1992-03

Multilevel security in data compression and restricted character set translation

Tsai, Chien C.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/23973

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

MULTILEVEL SECURITY WITH DATA COMPRESSION
AND
RESTRICTED CHARACTER SET TRANSLATION

by

Chien C. Tsai

March, 1992

Thesis Advisor: Chyan Yang

Approved for public release; distribution is unlimited

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b DECLASSIFICATION/DOWNGRADING SCHEDULE

3 DISTRIBUTION/AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited.

4 PERFORMING ORGANIZATION REPORT NUMBER(S)

5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION
Naval Postgraduate School

6b OFFICE SYMBOL *(If applicable)*
EC 55

7a NAME OF MONITORING ORGANIZATION
Naval Postgraduate School

6c ADDRESS *(City, State, and ZIP Code)*
Monterey, CA 93943-5000

7b ADDRESS *(City, State, and ZIP Code)*
Monterey, CA 93943-5000

8a NAME OF FUNDING/SPONSORING ORGANIZATION

8b OFFICE SYMBOL *(If applicable)*

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c ADDRESS *(City, State, and ZIP Code)*

10 SOURCE OF FUNDING NUMBERS

Program Element No	Project No	Task No	Work Unit Accession Number

11 TITLE *(Include Security Classification)*
MULTILEVEL SECURITY WITH DATA COMPRESSION AND RESTRICTED CHARACTERS SET TRANSLATION

12 PERSONAL AUTHOR(S) Chien C. Tsai

13a TYPE OF REPORT
Master's Thesis

13b TIME COVERED
From To

14 DATE OF REPORT *(year, month, day)*
1992, March

15 PAGE COUNT
86

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES

FIELD	GROUP	SUBGROUP

18 SUBJECT TERMS *(continue on reverse if necessary and identify by block number)*
Master key; LZSS; Shannon-Fano coding; Expansion Ratio; Recovery.

19 ABSTRACT *(continue on reverse if necessary and identify by block number)*

Multilevel military communication security can be implemented with the notion of masterkeys. The naval message traffic is transmitted with restricted character set and optionally the files are compressed. Both character translation and data compression can be used as add-on data encryption. A masterkey is constructed from a set of service keys from which the masterkey is allowed to access. This thesis presents the principles of multilevel security with restricted character translation, data compression, and masterkey implementation.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a NAME OF RESPONSIBLE INDIVIDUAL
YANG, Chyan

22b TELEPHONE *(Include Area code)*
408/6462266

22c OFFICE SYMBOL
EC/Ya

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Multilevel Security in Data Compression
and
Restricted Character Set Translation

by

Chien C. Tsai
Lieutenant Colonel, Taiwan Republic of China Army
B.S., Chinese Military Academy

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

ABSTRACT

Multilevel military communication security can be implemented with the notion of masterkeys. Naval message traffic is transmitted with restricted character set and optionally files are compressed. Both character translation and data compression can be used as add-on data encryption. A masterkey is constructed from a set of service keys from which masterkey is allowed to access. This thesis presents the principles of multilevel security with restricted character translation, data compression, and masterkey implementation.

TABLE OF CONTENTS

I. INTRODUCTION

Multilevel security is a familiar scheme of classification in the national security hierarchy. It may partition subjects in levels of *clearance* and divide objects into levels of *classification* [Ref. 1]. This thesis reports one implementation that supports multilevel security with a **Master Key** [Ref. 2] and is suitable for naval message traffic by **data compression** and **character translation.** Data compression reduces original data package size for better storage and transmission capacity, while character translation converts each byte of an input file to a restricted character set since naval message traffic uses a restricted character set.

A shore-based system includes a large database which consists of relational tables of ASCII data in a commercial RDBMS (Relational Database Management System) as well as associated ASCII text and binary (graphic) files. Packages of data are prepared from the database for subsequent delivery to remote systems via floppy disks, an electronic network, or via standard naval message traffic. Each prepared data package consists of a combination of ASCII and binary files grouped together in the standard hierarchical file storage structure of the host system. After either physical or electronic delivery of a data package, it resides within the file storage of the remote system.

The processes of data compression is followed by data encryption prior to passing the processed file on to the character translation process. The entire scenario is then

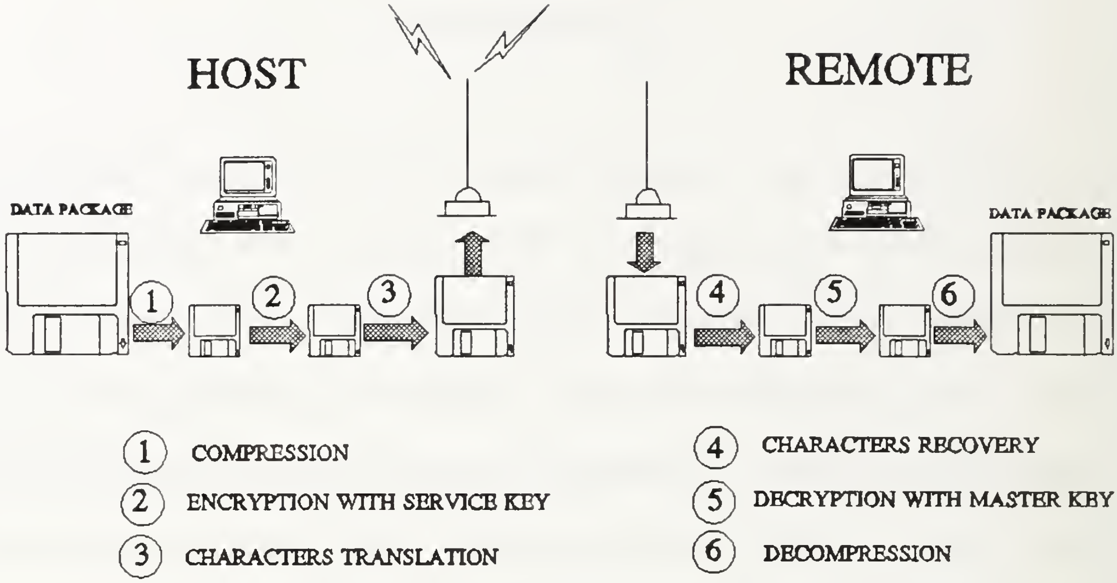


Figure 1. The scenario of data package transmission on Naval message traffic.

repeated in the reverse direction at the remote site as shown in Figure 1. A data compression algorithm is investigated based on its conflicting efficiencies of CPU time and compression ratio. When urgency does not dictate action, the compression ratio is more important than execution time. Data encryption is an optional requirement that allows different encryption schemes to be chosen. Without a hardware encryption chip available to us, DES (Data Encryption Standard) [Ref. 3] has been implemented by software that incorporates the Master Key routines. As an unclassified research, this thesis uses the UNIX crypt() routine for illustration.

Though designed for different purpose, all three methods used in this research contribute to data security: data compression, data encryption, and character translation. When all methods are employed, the overall data security is greatly enhanced since the probability of decoding by adversaries is the product of probabilities in breaking each process individually.

In Chapter II, data compression techniques are briefly overviewed. Chapter III discusses the incorporation of Master Keys for supporting multilevel security systems and data encryption. Chapter IV introduces the algorithm for character translation. The concluding remarks are given in Chapter V.

II. A CASCADING DATA COMPRESSION TECHNIQUE

Regardless of the compression algorithm used [Ref. 4, 5, 6, 7, 8], a compressed file can actually provide the first level of encryption since the compressed file consists of random bit patterns and mostly non-printable characters. Because all compression algorithms do not have byte-for-byte correspondence between the source file and the compressed file, each byte of the compressed file can not be reversed back to original bit pattern unless the compression algorithm is known.

The data compression software we use for implementing multilevel security access control and restricted character translation is the "ZIP" program written by Richard B. Wales et.al [Ref. 9]. The program includes three compression routines that may be chosen by the user. Each routine implements a different compression technique. In this chapter we overview the "Implode" routine. Implode is the best of the three algorithms and is known to be one of the fastest and most powerful schemes for data compression in terms of execution speed and compression ratio. The compression algorithm of "Implode" is actually a combination of two distinct algorithms. The first algorithm is OPM/L (Original Pointer Macro restricted to Left pointers) compression scheme which compresses repeated byte sequences using a sliding dictionary (window). The second algorithm is Shannon-Fano coding which uses multiple variable-length binary encoding of various parts of the OPM/L output.

A. THE OPM/L COMPRESSION ALGORITHM DEVELOPMENT

An OPM/L data compression scheme called LZ77 was first suggested by Ziv and Lempel [Ref. 6]. A slightly modified version of this scheme which improves the compression ratios for a wide range of texts, developed by Storer and Szymanski, is called LZSS [Ref. 10] and has fast decoding and requires comparatively little memory for coding and decoding.

An OPM/L scheme replaces a substring in a text with a pointer to a previous (left) occurrence of the substring in the text. The pointer represents the position and size of the sub-string in the original text. These restrictions make fast single-pass decoding straightforward. The LZ77 scheme restricts the reach of the pointer to approximately the previous N characters, effectively creating a "window" of N characters which are used as a sliding dictionary. Pointers are chosen using a greedy (seeking for longest match) algorithm which permits single-pass encoding. Therefore a LZ77 encoder is parameterized by N, the size of the "window", and F, the maximum length of a substring that may be replaced by a pointer. Encoding of the input string proceeds from left to right. At each step of the encoding a section of the input text is available in a window of N characters. Of these, the first N - F characters have already been encoded and the last F characters are the "lookahead buffer". For example, if the string S = abcabcbacbababcabc..... is being encoded with the parameters N = 11, F = 4 and character 12 is to be encoded next, the window is as shown in Figure 2.

Initially the first N - F characters of the window are (arbitrarily) blanks, and the first F characters of the text are loaded into the lookahead buffer. The already encoded

5	6	7	8	9	10	11	12	13	14	15
b	c	b	a	c	b	a	b	a	b	c

already encoded | lookahead buffer

Figure 2. LZ77 encoding string *S* with N = 11, F = 4.

part of the window is searched to find the longest match for the lookahead buffer, but obviously it can not be the lookahead buffer itself. In the example, the longest match for the "babc" is "bab", which starts at character 10. The longest match is then encoded into a triple <i, j, a>, where i is the offset of the longest match from the lookahead buffer, j is the length of the match, and a is the first character which did not match the substring in the window. In this example, the output triple would be <2, 3, 'c'>. The window is then shifted right j + 1 characters, ready for another coding step. Decoding is very simple and fast. The decoder maintains a window in the same way as the encoder, but instead of searching for a match in the window it uses the triple given by the encoder.

The main disadvantage of LZ77 is that, a straightforward implementation can require up to (N - F)·F characters comparisons, typically on the order of several thousands. The performance of different compression schemes with the parameters of speed and memory is listed in Table I.

An improved technique for reducing the time for compression was introduced by T.C.Bell [Ref. 4] since time is the only point when LZ77 or LZSS techniques fall short of other algorithms that are shown in Table I. The algorithm developed by Bell is the

Table I. Performance of different compression schemes.

COMPRESSION SCHEME	SPEED (bytes per second)		MEMORY (K bytes)	
	Encod	Decode	Encode	Decode
LZSS N=8192	18	13,600	8	8
LZSS N=2048	52	10,900	8	2
LZ77 N=8192	24	15,200	8	8
LZ78	5300	10,060	350	135
LZW	5700	8,400	48	12
ARITHMETIC	-	-	32 to 1400	32 to 1400
ADAPT. HUFF.	990	1,300	8	8

"Binary Tree Algorithm" that searches for the longest match for a string. Consider the same string *S* at page 5 with the same parameters N = 11 and F = 4, and coding is up to character 12 as shown in Figure 3.

5	6	7	8	9	10	0	1	2	3	4
b	c	b	a	c	b	a	b	a	b	c

| lookahead buffer |

Figure 3. Encoding window of Binary Tree Algorithm.

The lookahead buffer is defined as L = x_1 = babc and x_5 = bcba, x_6 = cbac, x_7 = bacb, x_8 = acba, x_9 = cbab, x_{10} = baba, x_0 = abab. By inspection, the longest match is x_{10} with vector (l, x) = (10, 3) where 10 is the position of the match string start and 3 is the characters that match the lookahead buffer.

The binary search algorithm start with sorting the x_5, x_6,..., x_0 with Literal order. So we have:

x_0	x_8	x_{10}	L	x_7	x_5	x_9	x_6
abab	acba	baba	babc	bacb	bcba	cbab	cbac

The longest match for L should be found at the beginning of x_{10} or x_7. This happened because these two strings are Literal adjacent to the lookahead buffer L and are the two candidates for the longest match.

The basic construction of the tree is that for any node x_i all nodes in its left subtree are Literal less than x_i and all nodes in its right subtree are Literal greater than x_i. Therefore the tree is constructed starting with x_5, x_6, x_7, ..., x_{10}, x_0, L and then x_{10}, and x_7 appear on the path to L as shown in Figure 4, where the algorithm for encoding and binary tree searching will be discussed in Section II. C.

B. SHANNON-FANO CODING ALGORITHM

The Shannon-Fano technique [Ref. 15] has as an advantage its simplicity. The code is constructed as follows. The source messages a_i and their probabilities $p(a_i)$ are listed in order of nonincreasing probability. This list is then divided in such a way as to form two groups of as nearly equal total probabilities as possible. Each message in the first group receives 0 as the first digit of its codeword; the messages in the second half have

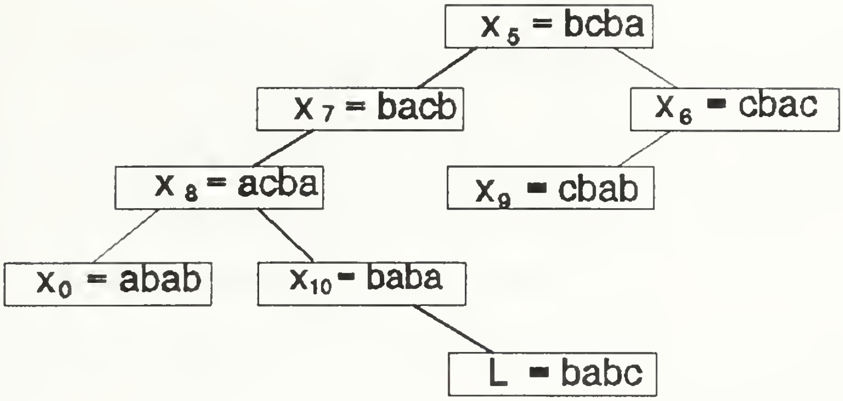


Figure 4. Lexicographically tree structure.

codewords beginning with 1. Each of these groups is then divided according to the same criterion, and additional code digits are appended. The process is continued until each subset contains only one message.

Figure 5 shows the application of Shannon-Fano algorithm to a specific probability distribution. The length of each codeword is equal to $-\log_2 p(a_i)$. This is true as long as it is possible to divide the list into subgroups of exactly equal probability. When this is not possible, some codewords may be of length $-\log_2 p(a_i) + 1$. The Shannon-Fano algorithm yields an average codeword length S that satisfies $H \leq S \leq H + 1$, where H is the entropy of the source. The Shannon-Fano code for the ensemble "aa bbb cccc ddddd eeeeee fffffff gggggggg" is shown in Figure 6.

CHARACTER	PROBABILITY	ENCODED	
a_1	1/2	0	step 1
a_2	1/4	10	step 2
a_3	1/8	110	step 3
a_4	1/16	1110	step 4
a_5	1/32	11110	step 5
a_6	1/32	11111	

Figure 5. A Shannon-Fano code.

CHARACTER	PROBABILITY	ENCODED	
g	8/40	00	step 2
f	7/40	010	step 3
e	6/40	011	step 1
d	5/40	100	step 5
space	5/40	101	step 4
c	4/40	110	step 6
b	3/40	1110	step 7
a	2/40	1111	

Figure 6. A Shannon-Fano code example.

C. SOFTWARE IMPLEMENTATION

The cascading of OPM/L and Shannon-Fano scheme in the Imploding algorithm can use a 4K (N = 4096) or 8K (N = 8192) sliding dictionary size. The dictionary size used can be determined by bit 1 in the general purpose flag word of "Local file header" [Ref. 11].

The Shannon-Fano trees are stored at the beginning of the compressed package. The number of trees stored is defined by bit 2 in the general purpose flag word. If 3 trees were stored, the first tree represents the encoding of the literal characters, the second tree represents the encoding of the Length information, the third represents the encoding of the distance information. When 2 Shannon-Fano trees are stored, the Length tree is stored first, followed by the distance tree.

The Literal Shannon-Fano tree, if presented, is used to represent the entire ASCII character set, and contains 256 values. This tree is used to compress any data not compressed by the sliding dictionary algorithm. When this tree is presented, the Minimum Match Length (MML) for the sliding dictionary is 3. If this tree is not presented, the MML is 2. The Length Shannon-Fano tree is used to compress the Length part of the (length, distance) pairs from the sliding dictionary output. The Length tree contains 64 values, ranging from the MML to MML + 63. The distance Shannon-Fano tree is used to compress the Distance part of the (length, distance) pairs from the sliding dictionary output. The Distance tree contains 64 values, ranging from 0 to 63, representing the upper 6 bits of the distance value. The distance values themselves will be between 0 and the sliding dictionary size, either 4K or 8K.

The Shannon-Fano trees themselves are stored in a compressed format. It can be constructed from the bit lengths using the following algorithm :

1) Sort the Bit Lengths in ascending order, while retaining the order of the original lengths stored in the file.

2) Generate the Shannon-Fano trees use the routine in Figure 7.

```
Code <- 0
CodeIncrement <- 0
LastBitLength <- 0
i <- number of Shannon-Fano codes - 1  (either 255 or 63)

loop while i ≥ 0
    Code = Code + CodeIncrement
    if BitLength(i) <> LastBitLength then
        LastBitLength = BitLength(i)
        CodeIncrement = 1 Shifted left (16-LastBitLength)
    ShannonCode(i) = Code
    i <- i - 1
end loop
```

Figure 7. Algorithm for generating Shannon-Fano trees.

3) Reverse the order of all the bits in the ShannonCode() vector, so that the most significant bit becomes the least significant bit. For example, the value 0x1234 (hex) would become 0x2C48 (hex).

4) Restore the order of Shannon-Fano codes as originally stored within the file.

Let's give an example which will show the encoding of a Shannon-Fano tree of size of 8. Notice that the actual Shannon-Fano trees used for Imploding are either 64 or 256 entries in size.

For example give : 0x02, 0x42, 0x01, 0x13.

The first byte indicates 3 values in this table. Decoding the bytes:

0x42 = 5 codes of 3 bits long

0x01 = 1 code of 2 bits long

0x13 = 2 codes of 4 bits long

This would generate the original bit length array of :

(3, 3, 3, 3, 3, 2, 4, 4)

There are 8 codes in this table for the values 0 through 7. Using the algorithm to obtain the Shannon-Fano codes step by step will produce the result as in Figure 8.

			Reversed	Order	Original
Val	Sorted	Constructed Code	Value	Restored	Length
0:	2	1100000000000000	11	101	3
1:	3	1010000000000000	101	001	3
2:	3	1000000000000000	001	110	3
3:	3	0110000000000000	110	010	3
4:	3	0100000000000000	010	100	3
5:	3	0010000000000000	100	11	2
6:	4	0001000000000000	1000	1000	4
7:	4	0000000000000000	0000	0000	4

Figure 8. Decoding steps of Shannon-Fano scheme.

The values in the 'Val', 'Order Restored' and 'Original Length' columns now represent the Shannon-Fano encoding tree that can be used for decoding the Shannon-Fano encoded data. The compressed data stream begins immediately after the compressed Shannon-Fano data. The compressed data stream can be interpreted by the algorithm shown in Figure 9.

Since pass two (Shannon-Fano tree) depends on a statistical analysis of the entire

```
loop until done
    read 1 bit from input stream.

    if this bit is non-zero then  (encoded data is
                                      literal data)
        if Literal Shannon-Fano tree is present
           read and decode character using Literal
           Shannon-Fano tree.
        otherwise
           read 8 bits from input stream.
        copy character to the output stream.
    otherwise  (encoded data is sliding dictionary match)
        if 8K dictionary size
           read 7 bits for offset Distance (lower 7 bits
                                             of offset).
        otherwise
           read 6 bits for offset Distance (lower 6 bits
                                             of offset).


        using the Distance Shannon-Fano tree, read and
        decode the upper 6 bits of the Distance value.

        using the Length Shannon-Fano tree, read and
        decode the Length value.

        Length <- Length + Minimum Match Length

        if Length = 63 + Minimum Match Length
           read 8 bits from the input stream,
           add this value to Length.

        move backwards Distance+1 bytes in the output
        stream, and copy Length characters from this
        position to the output stream. (if this position
        is before the start of the output stream, then
        assume that all the data before the start of the
        output stream is filled with zeros).
end loop
```

Figure 9. The OPM/L decoding algorithm.

output of pass one (OPM/L compression), the output of pass one is saved in a temporary file and re-read for pass two. Imploding is thus a two-pass algorithm.

Table II is a result of comparing different compression algorithms with different

Table II. Comparison of compression ratios.

		COMPRESSION ALGORITHM			
FILE	SIZE	DYNAM. LZW	LZSS	ADPTIVE HUFFMAN	CASCADING OPM/L
TXT	24969	46.6%	47.1%	58.9%	37.5%
WPR	25195	48.5%	49.6%	50.6%	40.4%
CPG	17325	39.5%	32.9%	58.6%	26.1%
EXE	24630	68.3%	55.9%	77.0%	53.6%
PAK	76644	55.2%	52.5%	70.1%	39.3%

TXT : Text file. WPR : Wordperfect file.
CPG : C source file. EXE : Executing binary file.
PAK : Combination of 90% text and 10% image binary file.

data type files in compression ratio (% of original file size). Clearly, the cascading of OPM/L and Shannon-Fano scheme has a far better performance than others. The EXE file exhibits a higher ratio because of the poor compression performance for binary files [Ref. 12], Run-length encoding may take advantage of long string of binary images which is not the main subject of this research. A complete comparison of data compression efficiency in terms of compression ratio and execution time can be found in Jung [Ref. 12].

III. DATA ENCRYPTION WITH MASTER KEY IMPLEMENTATION

When a data package is transmitted to remote systems, compressed or not, if encryption is requested by user, the access control of the encrypted package is then of principal concern. When several parties require shared access to a secure data package, it is convenient to partition the packages into several classes (multilevel) and encrypt each class individually. A key management problem can be avoided by providing a Master Key to permit access to the required classes. In this chapter, the Master Key scheme is introduced and implemented to support multilevel security.

A. THE MASTER KEY SCHEME

1. Master Key Systems

The brief overview of Master Keys in this section is based on [Ref. 2] which is an improvement to [Ref. 13]. A Master Key is a compact representation for a subset of the service keys. In the following discussion, the '$\leq$' indicates a partial order subordinating relation. Each S_i is assigned a service key SK_i. If $S_i \leq S_j$ then service (an object) S_i is subordinated to S_j and access to S_j guarantees access to S_i.

Each service is assigned a small prime p_i but no primes are assigned to the Central Authority (CA) or Master Keys user. Let

$$T = \prod_{n=1}^{N} p_n \ .$$

For each service a number u_i is defined as

$$u_i = \prod_{S_n \leq S_i} p_n \text{ ,}$$

and the service key is defined as

$$SK_i = K_0^{\frac{T}{u_i}}$$

K_0 is a random key number chosen by the central authority. The Master Keys can be made by the following mechanism. First, v_j is computed as

$$v_j = \prod_{SK_n \leq MK_j} p_n$$

where the set

$$\{ SK_i \leq MK_j \}$$

consists of all the keys for services accessible with Master Key MK_j. The Master Key is defined as

$$MK_j = K_0^{\frac{T}{v_j}} \text{ .}$$

The computation of a service key from a Master Key is then

$$SK_i = K_0^{\frac{T}{u_i}} = (K_0^{\frac{T}{v_j}})^{\frac{v_j}{u_i}} = MK_j^{\frac{v_j}{u_i}}, \qquad iff \;\; SK_i \le MK_j \tag{1}$$

Note that in this chapter, the arithmetic is performed in (mod M) for some integer M. Values are operated in the ring of integers (0, M - 1) where M is defined by

$$M = p_1 \cdot p_2$$

for some large primes p_1 and p_2 similar to the RSA algorithm [Ref. 14].

2. Flexibility of Master Key algorithm

Computation of a service key is feasible if and only if $SK_i \le MK_j$. If $SK_i \le MK_j$, then (by definition) all primes in u_i must be included in v_j, thus, u_i divides v_j. $(MK_j)^{vj/ui}$ is easily computed as in equation (1) since v_j/u_i results in an integer.

a. Prohibition of non-MasterKey intrusion

When $SK_i \not\le MK_j$, the access is denied as follows. Let

$$u_i = \alpha \cdot p_i, \qquad \therefore \; MK_j^{\frac{v_j}{u_i}} = [(MK_j)^{\frac{v_j}{p_i}}]^{\frac{1}{\alpha}}. \tag{2}$$

Where p_i does not divide v_j, and the p_i^{th} root of MK_j must be computed. But computing the r^{th} roots mod M for $r > 1$ is believed to be as difficult as factoring M [Ref. 3]. So when p_i does not divide v_j, $MK_j^{vj/pi}$ cannot be computed if the factors of the modulus are unknown. This prohibits the unauthorized access when M is large.

b. Prohibition of grouped intrusion

The Master Key is also secure against illicit cooperation where a group of people may have sufficient information to do things none of them are capable of individually. A sufficient condition is that no group of Master Keys can be used to gain access to additional services. That is, from a group of Master Keys we cannot create a key MK, such that $SK_i \leq MK$, if none of the keys in the group have access to service S_i. This has been proven in [Ref. 2] since p_i is not a factor of v_j in any Master Key of the group.

c. Expansion capability

[Ref. 2] also proved that it is possible to add services to the system, without affecting existing keys, provided that a new addition is not subordinate to any existing service. Hence a new service added will introduce the equation to compute SK_i as

$$SK_{N+1} = (K_0')^{\frac{T'}{u_{N+1}}} \qquad (3)$$

Where

$$K_0' = (K_0)^{\frac{1}{p_{N+1}}},$$

$$T' = T \cdot p_{N+1} ,$$

$$u_{N+1} = \prod_{S_n \leq S_{N+1}} p_n .$$

and SK_{N+1} is the new service added.

Although the keys are unchanged by the substitution of T' and K_0', but this manner bring up the new problem that number of services to be added is constrained by the relatively primes to M, for instance, if $M = p_1 * p_2$, then the maximum number of service key that can be expanded is 2. In addition, since the new prime p_{N+1} is not a factor of v_j for any of the existing keys, new Master Keys have to be redistributed by the CA to accommodate the $\leq$ relationships.

We notice that T value (products of all primes) is the burden which makes system expansion inflexible. T must be fixed or all key numbers have to change accordingly since T/u_i, or T/v_j provides the power of SK_i, MK_j. In our experiment, we introduce another method for Master Key system expansion. Numbers of individual services are assigned in the beginning when a system was built according to future expansion consideration. These services originally can be either $\leq$ or doesn't $\leq$ to any service or Master Key, but primes are assigned to them as usual service key. Thus, T value will not be affected when any of these services is assigned to be a new service key, and the new service key can be inserted (or activated) in between two existed service keys or under any Master Key as required, the only value has to be modified is the u_i or/and related v_j when insertion takes place.

B. EXAMPLE OF A MULTILEVEL SECURITY MODEL

To implement the concept of Master key scheme on compressed/encrypted data package, we constructed a multilevel hierarchy of 70 services as illustrated in Figure 10.

Each service can be treated as an encrypted data package. To access (decrypt) an encrypted package S_i from S_j or by MK_j, the subordinating relationship either $S_i \leq S_j$ or $S_i \leq MK_j$ must be satisfied respectively.

In Figure 10 the $\leq$ relationships among services are shown by covering over the inferior one in vertical order. For example, the leftmost column shows that $S_{08} < S_{09}$, $S_{07} < S_{08}$ ..etc. Therefore, a Master Key that can access a service S_i can also access all the services inferior to S_i. Thus, if a Master key MK $\geq$ SK_i, then this MK can access SK_{i-1}, SK_{i-2}, ... as well, provided that SK_{i-1}, SK_{i-2}, ... are connected and been covered. Consequently, this Master key is called MK_i.

Additionally, there are eight other Master keys (MK_j, j = 0,1,...,7) shown as shaded arrows in Figure 10. Each MK_j covers or is superior to services connected by a line. For example, MK_0, has a line going through S_{09}, S_{18}, S_{27}, S_{36}, S_{45}, S_{54}, S_{63} and therefore $SK_{09} \leq MK_0$, $SK_{18} \leq MK_0$,....., $SK_{63} \leq MK_0$. Table A in Appendix B lists the corresponding prime number assignments for Figure 10. For instance, the prime number for S_{10} is 349.

1. Example of access control

As an instructive example, let's arbitrarily pick S_{35} as an encrypted data package to access. Figure 10 shows $SK_{35} \leq MK_0$ ($SK_{35} \leq SK_{36} \leq MK_0$), $SK_{35} \leq MK_3$, $S_{35} \leq S_{36}$,...etc. Hence, there are 8 Master Keys that can access S_{35}:

$$\{ MK_0,\ MK_3,\ MK_5,\ MK_{35},\ MK_{36},\ MK_{37},\ MK_{38},\ MK_{39} \} \qquad (4)$$

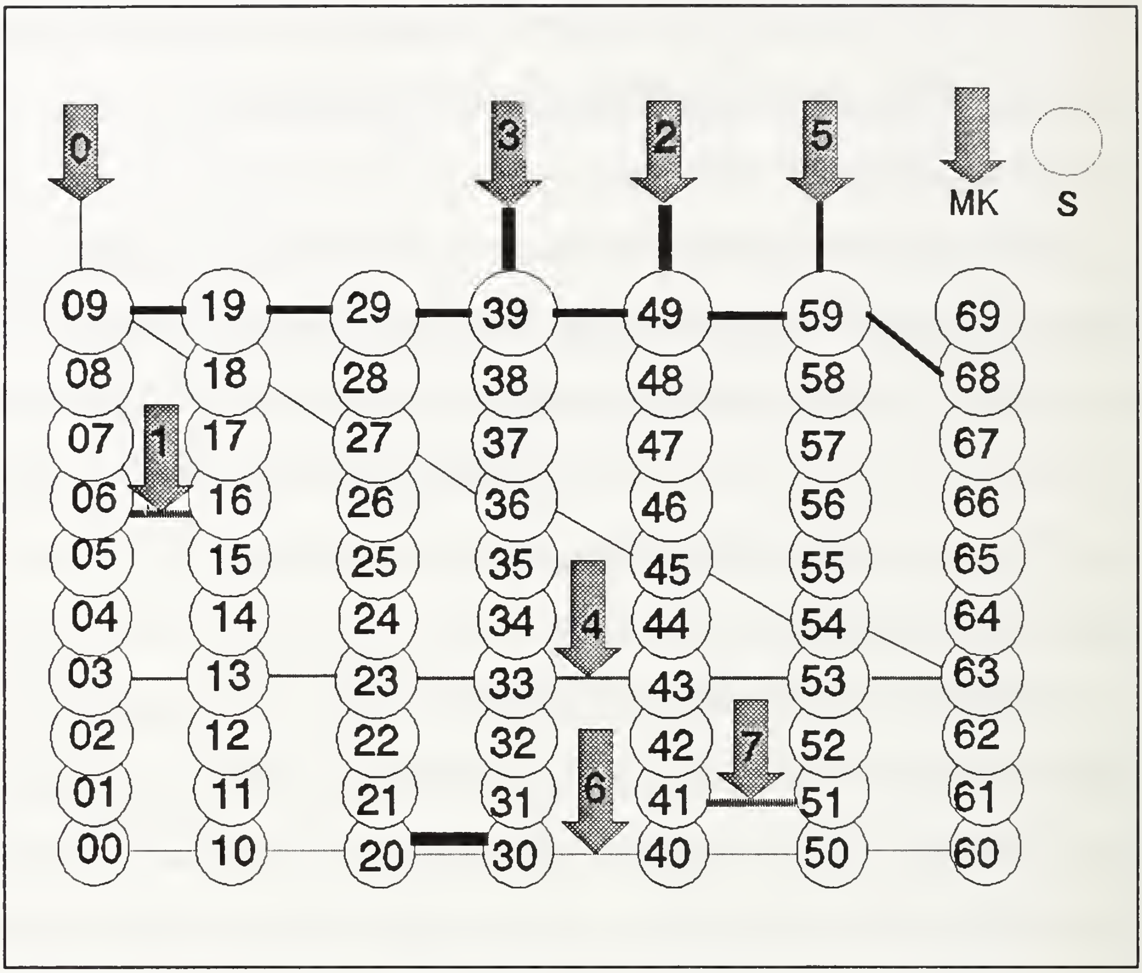


Figure 10. A multilevel services hierarchical model.

The number u_{35} is computed as

$$u_{35} = \prod_{i=30}^{35} p_i = 229 \cdot 227 \cdot 223 \cdot 211 \cdot 199 \cdot 197 \qquad (5)$$

Two parameters are needed for key number computation : First, M = 2147483641L is arbitrarily chosen such that M is close to 2^{31} and $M >> (p_0 \cdot p_1)$. This M can be supported by a 32-bit unsigned "long" integer in C. Next, a randomly picked K_0 = 1992 is used. Now the service key for S_{35} is

$$SK_{35} = \mathrm{mod}(K_0^{\prod_{n=0}^{69} p_{n \neq i}}) \qquad \mathit{for}\ i = 30,\ 31,\ 32,\ 33,\ 34,\ 35. \tag{6}$$

$$= 1989952527L$$

Table B in Appendix B lists all the service keys. We now show how the Master Key MK_3 can access S_{35} by making SK_{35} from MK_3. The value of v_3 has to be computed first.

$$v_3 = (\prod_{j=30}^{39} p_j) \cdot p_{20} = 179 \cdot 181 \cdot \ldots \cdot 229 \cdot 281 \tag{7}$$

Therefore,

$$MK_3 = 1992^{\prod_{n=0}^{69} p_n} \qquad \mathit{for}\ n \neq j,\ j = 30,\ 31, \ldots,\ 39,\ 20. \tag{8}$$

$$= 0002255128L$$

With MK_3 one can derive SK_{35}:

$$\begin{aligned} MK_3^{\frac{v_3}{u_{35}}} &= \mathrm{mod}((2255128)^{\frac{p_{20} \cdot p_{30} \cdot p_{31} \cdot \ldots \cdot p_{39}}{p_{30} \cdot p_{31} \cdot \ldots \cdot p_{35}}}) \\ &= \mathrm{mod}((2255128)^{p_{20} \cdot p_{36} \cdot p_{37} \cdot p_{38} \cdot p_{39}}) \\ &= \mathrm{mod}((2255128)^{281 \cdot 193 \cdot 191 \cdot 181 \cdot 179}) \\ &= 1989952527L = SK_{35} \end{aligned} \tag{9}$$

In other words, MK_3 can access service S_{35}.

2. Example of intrusive prevention

On the other hand, let's see whether MK_7 (it is not in set (4)) can access S_{35}.

Since

$$\begin{aligned} v_7 &= p_{40} \cdot p_{41} \cdot p_{50} \cdot p_{51} \\ &= 173 \cdot 167 \cdot 113 \cdot 109 = 355850447L \end{aligned} \tag{10}$$

and

$$\begin{aligned} MK_7 &= \mathrm{mod}(1992^{\prod_{n=0}^{69} p_n}) \quad for\ n \neq j,\ j = 40,\ 41,\ 50,\ 51. \\ &= 1479772666L \end{aligned}$$

To make the service key SK_{35} from MK_7 one may try the following.

$$MK_7^{\frac{v_7}{u_{35}}} = \mathrm{mod}((1479772666)^{\frac{p_{40} \cdot p_{41} \cdot p_{50} \cdot p_{51}}{p_{30} \cdot p_{31} \cdot \cdots \cdot p_{35}}}) \tag{11}$$

Let

$$\alpha = p_{30} \cdot p_{31} \cdot p_{32} \cdot p_{33} \cdot p_{34}.$$

Equation (11) becomes

$$MK_7^{\frac{v_7}{u_{35}}} = \mathrm{mod}(((1479772666)^{\frac{p_{40} \cdot p_{41} \cdot p_{50} \cdot p_{51}}{p_{35}}})^{\frac{1}{a}}.$$

Since p_{35} does not divide into v_7, MK_7 can not access S_{35} and computing the p_{35}th root to factor p_{35} out will be difficult; never even mention the modulus number 2147483641L. With same computation model, one can show that all Master Keys in set (4) can access S_{35}, but none of the others can. It is also shown that a Master Keys constructed from a group of Master Keys which are not in set (4) access S_{35} either, since p_{35} (197) does not divide any v of them. This prevents the grouped intrusion.

C. SOFTWARE IMPLEMENTATION

Recall the scenario of the Naval message traffic described in Chapter I. If data encryption is required, it has to be done after data compression and before the characters set translation at the host system. In the software implementation (see Appendix A), user may provide the password (key) for encryption after data compression (at host system) or data recovery (at remote system). Given a correct key the program will encrypt or decrypt the file; otherwise the program assumes no data encryption.

Command line options allows a user to modify an encrypted file without explicitly decrypting it. But when using this option a user still has to provide the decryption key, The whole procedure is as follows:

[recovery] → decryption → modification → encryption → [translation].

In this case, the password for encryption procedure is automatically derived from file header of the decryption process. In the next two sections, we will discuss how passwords are verified in the Master Key access control environment and how data encryption is implemented. The C program listing can be found in Appendix A.

1. Master Key access control

Access control is divided into two steps: password conversion and key number computation.

```
first 2 numbers = first byte (of password) convert to
ASCII;
third number = second byte - 49 (ASCII);
fourth number = third byte - 52 (ASCII);
fifth number = fourth byte - 55 (ASCII);
sixth and seventh number = fifth and sixth byte;
eighth number = seventh byte - 58 (ASCII);
ninth number = eighth byte - 61 (ASCII);
tenth number = ninth byte - 64 (ASCII);
key number = conversion all ASCII number to type long;
identify number (u, or v) = last 2 bytes of password;
return key number;
```

Figure 11. Password conversion algorithm.

a. Key to Password Conversion

As specified in section III. B. To facilitate the friendly use by the users, passwords are used instead of the keys, The key to password conversion can be done in several ways. Here we present one way that is easily implemented. A key number is implemented as a 10-digit number since the chosen modulo number is 2147483641. Key number as well as the index number of *v* (*u*) are converted to a 11-character password by simply performing the shift and translation as shown in Figure 11. Hence, SK_{00}

(0015206469L) is converted to 'abii06nsy20'. In Appendix B, Table B, all Master Key numbers and passwords are listed.

At the host system, the user password is examined to avoid the data package being ruined by an invalid password. Verified password is then itself encrypted and embedded in the file header starting at the 4th byte. At the remote system, the received file header will be examined to see if it was a encrypted file. If it is encrypted, the 11 bytes starting at 4th byte in header will be used for service key number conversion. Meanwhile, the user must provide the password for key computation. In Appendix B, all Master Key numbers and passwords are listed.

b. Service key number computation

In using the Master Key, it is assumed that there is no password distributed electionically and the access is done by key number computation. The Master Key numbers are not necessary to be the same as service key numbers before computation. Therefore, different Master Key numbers may result same service key number based on a unique v_j.

Recall the discussions in sections A, and B, the modulus operation was taken in each arithmetic operation. While implementing in software, care has to be taken that modulus operation will not work if it is trying to mod a key number result as in (1), since multiple more prime numbers normally produce a large digits number. Without memory concatenation, the product result will be truncated and become useless before modulus process because the Floating Point Number System allows limited digits (Mantissa) representation, (e.g. IEEE-double precision has 53 bits [Ref. 16]). An example here is the *T* value by definition :

$$T = \prod_{n=0}^{69} p_n = 409 \cdot 401 \cdot 397 \cdot \ldots \cdot 37 \cdot 31 \qquad (12)$$

It result a number that has more than 70 digits and can not be easily represented. Moreover, since the products of primes will become the power of K_0, no Floating Point Number System can support such large value. To solve this problem, a procedure we called "wrapping" is implemented in software as listed in Appendix A. Modulus operation is now beginning at first k_0^p, and repeat in each multiplication until the end, to restrict each result in the range [0, 2147483640]. Now (1) becomes

$$\begin{aligned} SK_i &= \mathrm{mod}(k_0^{\frac{T}{u_i}}) \\ &= \mathrm{mod}(k_0^{\frac{\prod_{n=0}^{69} p_n}{\prod_{s_n \leq s_i} p_n}}) \\ &= \mathrm{mod}(k_0^{\prod_{s_n \leq s_i} p_n}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad (13) \\ &= \mathrm{mod}(k_0^{p_a \cdot p_b \cdot \ldots \cdot p_n}) \qquad \mathit{for}\ S_a, S_b, \ldots, S_n \nleq S_i \\ &= \mathrm{mod}(\mathrm{mod}((\ldots\mathrm{mod}((\mathrm{mod}(k_0^{p_a}))^{p_b})\ldots)^{p_n})) \end{aligned}$$

Furthermore, to prevent a large prime number as a power, any inner term in (13) can be divided into

$$\mathrm{mod}(m^{p}) = \mathrm{mod}(..\mathrm{mod}(\mathrm{mod}(m)_{1} \cdot m)_{2} .. \cdot m)_{p} \qquad (14)$$

Key number computation is implemented only at the remote system right after password conversion. If the Master Key number after computation is equal to the service key number, picked up from file header, then the program will decrypt with the password obtained from the file header. The algorithm for password verification and key number computation is shown in Figure 12.

2. Data encryption

Encryption on a compressed data package is an option to user. It may be specified at the command line when executing the software at the host system. If it is requested, the function encrypt() will process after password is verified. At remote system, program will automatically get the first 3 bytes in file header to check if it is an encrypted file and verify the key. Notice that encryption algorithms are varied from user to user, so are the password encryption of file header and key number conversion. They can be implemented in different schemes to meet the data security requirements, for example the DES. After all, the Master Key scheme will not be affected by different encryption schemes and it drives the access control.

In this experimentation we used the UNIX crypt() (the key generation part from "Makekey" has been modified to Master Key password conversion) routine for

```
loop:  request user enter password;
       if user password == header password: (the case of
                                         encrypted user)

          proceeding decryption;

       else if length of password != 11:
          if over three tries:
             recover original file and exit;
          else
             display error message then go to loop;

       else
          passw2num(user password);

       encrypted key number = passw2num(header password);
       initial for key computation;
       if Master Key user:
          get belonging service list matrix;
          loop to product of relate primes (v) times:

          keynum_mod();

       else (the case of superior service keys)
          loop to product of relate primes (u) times:
          keynum_mod();

       if computed key number == encrypted key number:
          proceeding decryption;

       else
          if over three tries:
             recover original file and exit;
          else
             display error message then go to loop;
```

Figure 12. Password verification algorithm.

from "Makekey" has been modified to Master Key password conversion) routine for illustration. It is a one-rotor machine encryption algorithm designed along the lines of Enigma but considerably trivialized, encryption and decryption uses the same key. Each included 3 bytes for encryption distinguish and 11 bytes for password (it is the keyword to encrypt too). Encrypted data packages must be decrypted before they can be decompressed. To decrypt a package, a shift operation is used to decrypt the 11-byte keyword for the encryption key which in turn decrypts the compressed data stream if the password has been verified.

IV. RESTRICT CHARACTER SET TRANSLATION

Since Naval message traffic uses the restricted character set listed in ntp3 annex C, the compressed and/or encrypted package has to be translated using this restricted character set. A restricted character set of N symbols can be represented as

$$C = \{ \alpha_1, \alpha_2, \ldots\ldots \alpha_N \} \quad where \quad N \leq 256. \qquad (15)$$

Without loss of generality, it is assumed that the restricted 45 (N) characters are in the contiguous decimal value interval [46, 90] (ASCII '.' to 'Z'). In a source package (without character translation) each input byte of 8 bits can assume $2^8 = 256$ various bit patterns and all patterns are equally likely to occur. When mapping a byte of 8-bit to one of the 45 restricted character, one may let 40 bit patterns uniquely map to corresponding 40 characters; the mapping of other 216 patterns have to use 2 characters each. On the average, the translated file is expanded to 185% of the original file since

$$\frac{40}{256} \times 1 + \frac{216}{256} \times 2 = 1.84375 \rightarrow 184.37\%.$$

The expansion ratio (85%) is unacceptably high therefore a source file must be translated in blocks of bits (< 8) when data compression efficiency is concerned. Since N = 45, 5 < $\log_2(45)$ < 6. Thus, the bit pattern to be translated could be blocks of either 5-bit or 6-bit depending on the efficiency of expansion ratio to be discussed below.

A translated character (8 bits) can represent a block of either 5-bit or 6-bit of input bit stream. This implies that an output byte (character) always starts with a '0' bit and the other 7 bits vary in 45 patterns. Hence, a shift operation is needed to output a byte in the desired range. Three basic translation methods are discussed as follows:

1). Scan input stream in 6-bit blocks. Since a 6-bit block may form 64 different patterns, with only 45 characters to map there are 44 lucky 6-bit pattern that can map to a single character in expression (15) whereas the rest of 20 patterns have to be translated in two combined characters $\alpha_{45}\alpha_i$, $i \neq 45$.

2). Scan input stream in 5-bit blocks. Since a 5-bit block may span 32 different patterns, with 45 characters to map there are 13 characters in the restricted character set unused.

3). This is an improvement to the second method. 13 unused characters are assigned to corresponding 6-bit patterns. It scans input stream in 6-bit blocks, examines the values and will translate 6-bit block whenever possible.

A. EXPANSION RATIOS

It can be shown that the first two methods are not as efficient as the third method. Without a prior knowledge of the source stream, it is reasonable to assume that all bit patterns are equally likely in the following discussions. Let b_1, b_2 be the number of bits used to encode a translated character (byte) and p_{b1}, p_{b2} be the corresponding probabilities of occurrences in translation. The expansion ratio η of method 1 is then

$$\eta = \frac{(8-b_1)}{b_1}(p_{b1}) + \frac{(8 \times 2-b_2)}{b_2}(1-p_{b1}) = 0.75 \rightarrow 75\% \qquad (16)$$

where

$$b_1 = 6, \quad p_{b1} = \frac{44}{64} = 0.688$$

$$b_2 = 6, \quad p_{b2} = 1 - p_{b1} = 0.312 \ .$$

In equation (16), the second term indicates that we expand from 6 bits to 2 bytes for the 20 unlucky 6-bit patterns. Similarly, we can compute the expansion ratio for the second method as

$$\eta = \frac{8-5}{5} = 0.6 \rightarrow 60\%.$$

For the method 3, $b_1 = 6$, $b_2 = 5$, thus

$$\eta = \frac{(8-6)}{6}\left(\frac{13}{64}\right) + \frac{(8-5)}{5}\left(\frac{51}{64}\right) = 0.546 \rightarrow 54.6\%. \qquad (17)$$

The third method provides the best of all three translation methods and is very close to set standard of 50%.

Variants of the method 3 can increase the probability of 6-bit block pattern translation. But, they are not as efficient as the method 3. For example, when N = 45, one may assign 16 patterns for 4-bit and 29 patterns for 6-bit. Two other variants are (1) 8 patterns for 3-bit and 37 patterns for 6-bit and (2) 4 patterns for 2-bit and 41 patterns for 6-bit. We calculate the corresponding expansion ratio for each case as follows:

- 29 patterns for 6-bit with others for 4-bit :

$$\eta = \left(\frac{8-6}{6}\right)\left(\frac{29}{64}\right) + \left(\frac{8-4}{4}\right)\left(\frac{35}{64}\right) = 0.698 \Rightarrow 69.8\%$$

- 37 patterns for 6-bit with others for 3-bit :

$$\eta = \left(\frac{8-6}{6}\right)\left(\frac{37}{64}\right) + \left(\frac{8-3}{3}\right)\left(\frac{27}{64}\right) = 0.896 \Rightarrow 89.6\%$$

- 41 patterns for 6-bit with others for 2-bit :

$$\eta = \left(\frac{8-6}{6}\right)\left(\frac{41}{64}\right) + \left(\frac{8-2}{2}\right)\left(\frac{23}{64}\right) = 1.292 \Rightarrow 129.2\%$$

All the results are worse than the method 3 because the second term in each calculation grows faster than the reduction of the corresponding probabilities. Moreover, it is impossible to translate more than 6 bits in each decision when N < 64. To generalize the η computation of method 3 for restricted character set of size N in the range [2, 256] the η can be calculated as follows:

$$\eta = \frac{(8-b_1)}{b_1}(p_{b_1}) + \frac{(8-b_2)}{b_2}(p_{b_2}) \qquad (18)$$

$$b_1 = \lceil \log_2(N) \rceil, \quad p_{b_1} = \frac{N - 2^{\lfloor \log_2(N) \rfloor}}{2^{\lceil \log_2(N) \rceil}}$$

$$b_2 = \lfloor \log_2(N) \rfloor, \quad p_{b_2} = 1 - p_{b_1}$$

Table III. A testing result of compression and translation.

FILE	SIZE(BYTE)	COMP	TRANSLATION		C&R	
			SIZE	η	SIZE	%ORG
TXT	24969	9361	38259	53.2%	14388	57.6%
WPR	25195	10189	38745	53.8%	15667	62.2%
CPG	17325	4517	26319	51.9%	6955	40.1%
EXE	24630	13204	37969	54.2%	20221	82.1%
PAK	76644	30129	117622	53.5%	46294	60.4%

TXT : Text file. WPR : Wordperfect file.
CPG : C source file. EXE : Execution file.
PAK : Combined of 90% ASCII and 10% binary file.
COMP : Compressed only.
η : Expansion ratio due to translation.
C&R : Compressed and Translated.
%ORG : % of original file size.

The equation (18) is similar to equation (17) except that it is now parameterized with N. Theoretically, by taking the expansion ratio 54.6% of equation (17) with an average compression ratio around 39.3% (derived from Table II, 'PAK' file), the compressed and translated file size would be about 39.3% × (1 + 54.6%) ≃ 60.77%

of original file size. Table III shows testing results of different type of files after compression and character set translation. The expansion ratios in 4th column agrees with the theoretical value in equation (17). The larger variance among testing results, however, is shown in the last column ("C&R %ORG" column) when both compression and translation are performed. This is due to the different file type benefits different compression ratio. It is interesting to note that PAK file (combination of 90% ASCII text and 10% image binary data which is a Data Representation Format specified by the Navy) size shown in 5th column becomes "C&R" 60.4% of original file, fairly close to theoretical ratio (60.77%) described above.

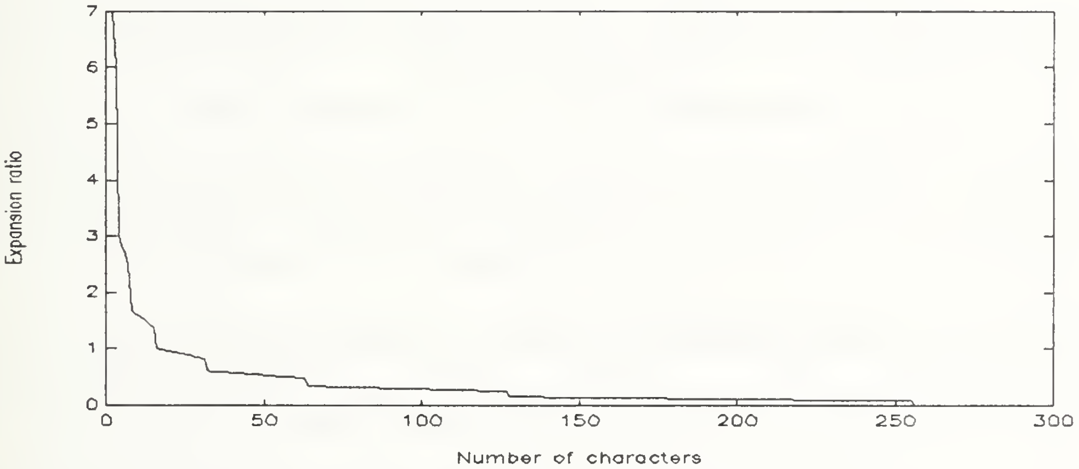


Figure 13. Expansion ratio with variable character numbers.

Figure 13 shows the plot of equation (18). The expansion ratio decline sharply before N reaches 16. Notice the stairlike steps when N is a power of 2, ratio curve doesn't continuously vary with N. This because the ratios analysis based on theoretically computes the translated bits corresponding probabilities. In software implementation, the bit-shift manipulation may improve the expansion ratio as will be explained in Section 3.

B. SOFTWARE IMPLEMENTATION CONSIDERATION

A C program (See Appendix A) based on method 3 was implemented. We now describe the algorithm that has been incorporated in a compressed/encrypted data package. The character set translation algorithm has two separate parts: translation (at host system) and recovery (at remote system).

1. Translation Algorithm

The translation algorithm scans the input stream, however, in 6-bit blocks before committing to a translation. We may assign 32 restricted characters ('.' through 'M') to decimal values interval [0, 31] for 5-bit blocks and the other 13 characters ('N' through 'Z') to the interval [32, 44] for 6-bit blocks. If the value of the 6-bit block is in the interval [32, 44], then the block is translated to the corresponding character. When the value is not in the range of [32, 44] then it is either in [0, 31] or in [45, 63]. The algorithm shifts one bit backward (unget) making the value reside in interval [0, 31] and translates the 5-bit block. In the following discussion let S denote the decimal interval [32, 44]. Refers to Figure 14, when the input string is coming from right to left, we observe the following:

Bit pattern 1 ('100110', the LSB is 0):

Decimal value = 38 $\in S$, translate the 6-bit block and output 38 + 46 ('T').

Bit pattern 2 ('001101'):

Decimal value = $13 \notin S$,

translate the 5-bit block of '00110' (shift 1 bit left), and output 6 + 46 ('4').

Bit pattern 3 ('111101'):

Decimal value = $61 \notin S$,

translate the 5-bit block of '11110' (shift 1 bit left), and output 30 + 46 ('L').

Bit pattern 4 ('100100'):

Decimal value = $36 \in S$, output 36 + 46 ('R').

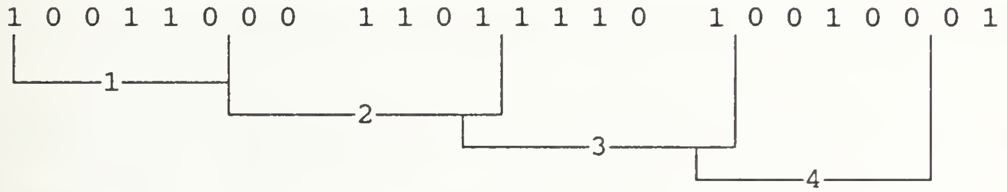


Figure 14. Bit pattern dissection of translation.

The displacement of 46 above is to map decimal values into the desired ASCII code range ['.', 'Z']. The output characters will be 'T4LR....' and leave the last 2 bits to be the MSB of the next 6-bit block. When the EOF or last byte of buffer is encountered, the remaining bits will be padded with 0s in LSB to form the last pattern. For the example in Figure 14, if '10010001' is the last input byte, then the last 6-bit pattern will be '010000' and translated to '01000' + 46 ('6'), the output is then 'T4LR6'. The inner loop of translation algorithm is listed in Figure 15.

```
loop: if the scanned 6-bit block is in interval
    [32, 44]:
        output (6-bit pattern+46);

      else
        unget 1 bit and output (5-bit pattern+46);

      rebuild bit pattern from remaining bits;
      if number of remaining bits >= 6:
        goto loop;

      else input next byte;
```

Figure 15. Translation algorithm.

2. Recovery Algorithm

The displacement of 46 made in translation has to be reset for each input character in recovery at the receiving hosts. If the value after reset is in *S*, an original 6-bit translated pattern is assumed and a 6-bit block is recovered; otherwise it maps to a 5-bit block. The output characters 'T4LR6'in Figure 14 will be recovered to the original bit string as shown in Figure 16.

Because the file before translation is byte-oriented, the recovery of the last input character should complete the last byte of original compressed/encrypted package. The inner loop of the recovery algorithm is listed in Figure 17.

Translation and recovery algorithms are two separate functions in the verify() routine of main() (listed in Appendix A). In the host system, character set translation is final step before transmission. The program takes each byte from the temporary file built by compression/encryption, and adds 3 bytes header in output file. Each of the 3-byte header, of course is within [46, 91] too. Moreover, the inner loop of translation function,

```
   'T' - 46       '4' - 46      'L' - 46       'R' - 46
 = 00100110     = 00000110    = 00011110     = 00100100
┌────┘    │     ┌───┘   │     ┌───┘   │     ┌────┘    │
1 0 0 1 1 0      0 0 1 1 0     1 1 1 1 0    1 0 0 1 0 0
└──────────────────┘ └─────────────────┘    └──────────
 Recover 1st byte          2nd byte          last byte..

   '6' - 46
  = 0001000
  ┌───┘   │
  0 1 0 0 0
  ──┘ └────
  ..   Ignore
```

Figure 16. Bit pattern dissection of recovery.

```
loop: input one byte from translated file;

        if (character-46) is in interval [0, 31]:
    skips 3 zero bits and recover 5 bits;

        else
           skips 2 zero bits and recover 6 bits;

        if number of bits in pattern buffer >= 8:
    output 1 byte and goto loop;

        rebuild bit pattern buffer from remaining
 input bits string;
```

Figure 17. Recovery algorithm.

trans() (See Figure 15), can be designed to incorporate the output buffer of compression/encryption for various compression algorithms. For instance, the variable buffer size in LZW algorithm requires variable loop index n ∈ [9, 13].

At the receiving system, recovery operation is proceeded first by examining the file header, similar to the encryption operation. The recovery function i.e. recov() shown in Figure 17, provides an option that allows user to modify/update the data package without change the translated format. The translation will be proceeded automatically after modification. Hence, the whole procedure becomes:

recovery → [decryption] → modification → [encryption] → translation.

C. IMPROVEMENT BY PATTERN REASSIGNMENT

In this section, each bit pattern corresponds to output character will be examined, and shows how we can further reduce the translation expansion ratio by suitable reassigns each of them.

1. Unused Patterns in Translation

The translation algorithms discussed in previous sections examines an input 6-bit block and translates it to either a 6 bits or a 5 bits block. Theoretically, when each input pattern is assumed to be equally likely occurred, having compressed and/or encrypted, as discussed in Section 2, the method 3 with expansion ratio 54.6% seems to be an optimized algorithm. Having enumerated all patterns, however, we can further reduce the expansion ratio to less than 50%.

The clue is that certain 5-bit patterns do not appear in practical translation procedure due to the 1 bit shift operation when the 6-bit value is not in S or interval [32, 44]. Figure 18 lists all 6-bit patterns with corresponding decimal values and output characters. Notice that in Figure 18 the 5-bit blocks in sets S_L and S_U exhibit redundancies and six patterns do not occur (values in interval [16, 21]), for instance, the first two 5-bit blocks in S_L are both '00000'. The 6 missing patterns are '10000', '10001', '10010', '10011', '10100', and '10101' with corresponding characters ' > ', '?', '@', 'A', 'B', 'C' respectively. In other words, when use method 3 in Section 2 we are translating 64 6-bit patterns to 39 (45 - 6) characters with 26 of them appear twice and leave 6 characters unused. The effort now is to translate patterns in S_L or S_U in 6-bit block by assigning unused characters to them. Reexamining Figure 18, we can verify that these missing characters in fact did not appear! That is, what we have done in previous section is restricted to 39 characters instead of 45. This observation could lead to the improvement of expansion ratio.

2. Characters Reassignment

We now consider how to use the 6 unused characters. These 6 unused characters may be assigned to the first six unique 6-bit blocks of S_U (from '101101' to '110010'). That is, we can assign these 6 characters to values in [45, 50]. By doing this, the characters originally assigned to interval [45, 50] (4 characters : 'D', 'E', 'F', and 'G') become unused. These four characters can be used to substitute another four 6-bit patterns, say [51, 54] (from '110011' to '110110'). Moreover, the characters 'H' and 'I' correspond to [51, 54] are reassigned to [55, 56]. This recursive characters reassignment may continue until value 57 was assigned when no more unused character. There are 6+4+2+1=13 characters has been reassigned. As shown in Figure 19 through appropriate displacement of +46 (within S_A), +30 (within S'), or +59 (within S_B) we can rearrange all output characters to be contiguous similar to that in ASCII code. The 6-bit patterns '100000' through '111001' (interval [32, 57]) is now assigned to characters

	6-BIT PATTERNS	5-BIT VALUE*	6-BIT VALUE	OUTPUT CHAR**.
	0 0 0 0 0 0	0		'.'
	0 0 0 0 0 1	0		'.'
	0 0 0 0 1 0	1		'/'
	.	.		.
S_L	.	.		.
	.	.		.
	.	.		.
	0 1 1 1 0 1	14		'<'
	0 1 1 1 1 0	15		'='
	0 1 1 1 1 1	15		'='
	1 0 0 0 0 0		32	'N'***
	1 0 0 0 0 1		33	'O'
	.		.	.
S	.		.	.
	.		.	.
	1 0 1 0 1 1		43	'Y'
	1 0 1 1 0 0		44	'Z'
	1 0 1 1 0 1	22		'D'
	1 0 1 1 1 0	23		'E'
	1 0 1 1 1 1	23		'E'
	.	.		.
S_U	.	.		.
	.	.		.
	1 1 1 1 0 1	30		'L'
	1 1 1 1 1 0	31		'M'
	1 1 1 1 1 1	31		'M'

* When unget 1 bit. ** Total 39 choices. *** 'N'= 32 + 46.

Figure 18. List of 6-bit patterns and corresponding output characters.

'>' through 'W'. All other 6-bit blocks (in S_A or S_B) still have to be translated in 5-bit block.

3. Expansion Ratio Improvement

The expansion ratio is improved because we increase the probability of translating 6-bit blocks and reduce that of 5-bit blocks. For all 64 possible 6-bit patterns,

	6-BIT PATTERNS	5-BIT VALUE	6-BIT VALUE	OUTPUT CHAR.
	.	.		.
S_A	.	.		.
	0 1 1 1 1 | 1	15		'='
	1 0 0 0 0 0		32	'>'
	1 0 0 0 0 1		33	'?'
	.		.	.
	.		.	.
	.		.	.
	1 0 1 0 1 1		43	'I'
	1 0 1 1 0 0		44	'J'
S'	1 0 1 1 0 1		45	'K'
	1 0 1 1 1 0		46	'L'
	1 0 1 1 1 1		47	'M'
	1 1 0 0 0 0		48	'N'
	1 1 0 0 0 1		49	'O'
	1 1 0 0 1 0		50	'P'
	1 1 0 0 1 1		51	'Q'
	1 1 0 1 0 0		52	'R'
	1 1 0 1 0 1		53	'S'
	1 1 0 1 1 0		54	'T'
	1 1 0 1 1 1		55	'U'
	1 1 1 0 0 0		56	'V'
	1 1 1 0 0 1		57	'W'
	1 1 1 0 1 | 0	29		'X'
	1 1 1 0 1 | 1	29		'X'
S_B	1 1 1 1 0 | 0	30		'Y'
	1 1 1 1 0 | 1	30		'Y'
	1 1 1 1 1 | 0	31		'Z'
	1 1 1 1 1 | 1	31		'Z'

Figure 19. List of 6-bit patterns with new assignments.

we now have 26 patterns that can be translated and 38 patterns have to be translated in 5-bit blocks. The overall expansion ratio is then :

$$\eta = \frac{8 - 6}{6}\left(\frac{26}{64}\right) + \frac{8 - 5}{5}\left(\frac{38}{64}\right) = 0.4917 \Rightarrow 49.17\%. \qquad (19)$$

This expansion ratio shows a 5.42% improvement over the expression (17) and achieves the specification set by the Navy. Hence, equation (18) can be rewritten as

$$\eta = (8 - b_1) \frac{p_{b_1}}{b_1} + (8 - b_2) \frac{p_{b_2}}{b_2},$$

$$b_1 = \lceil \log_2(N) \rceil, \quad p_{b1} = \frac{(N - 2^{\lfloor \log_2(N) \rfloor}) \times 2}{2^{\lceil \log_2(N) \rceil}}, \qquad (20)$$

$$b_2 = \lfloor \log_2(N) \rfloor, \quad p_{b2} = 1 - p_{b1}.$$

Where p_{b1} doubles the probability of equation (18) is the observed result from previous discussion. The number of characters assigned to translate each 6-bit block can always result in the same number of unused 5-bit characters; this is true when N varies. To compare with previous analysis, we can plot the expansion ratio vs number of characters for both equation (18) and (20) shown in Figure 20. Refers to Figure 14, the area between two curves is exactly the ratio improved by reassigning the unused characters. It seems that when N is in the range of [32, 64] the expansion ratios are quite satisfactory. Nevertheless, practical operation environment dictates the choice of N. For example, in order to accommodate a set of Morse code communication, the choice of N = 45 seems reasonable regardless of expansion ratio. When N < 16, it is not practical to perform the character translation since the expansion ratio can be as high as 700%. On the other hand, when N approaches to 256, there is no need for character translation since the source character set and the target character set are equal in size.

The modification of the translation program from that of method 3 to accommodate the observation made in this section is straightforward by adjusting the

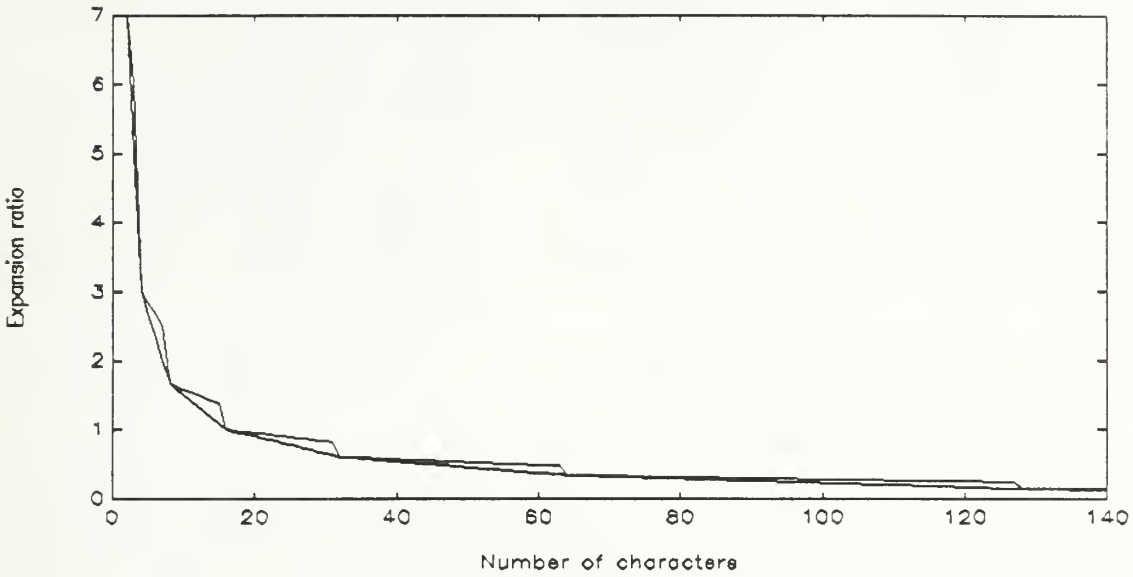


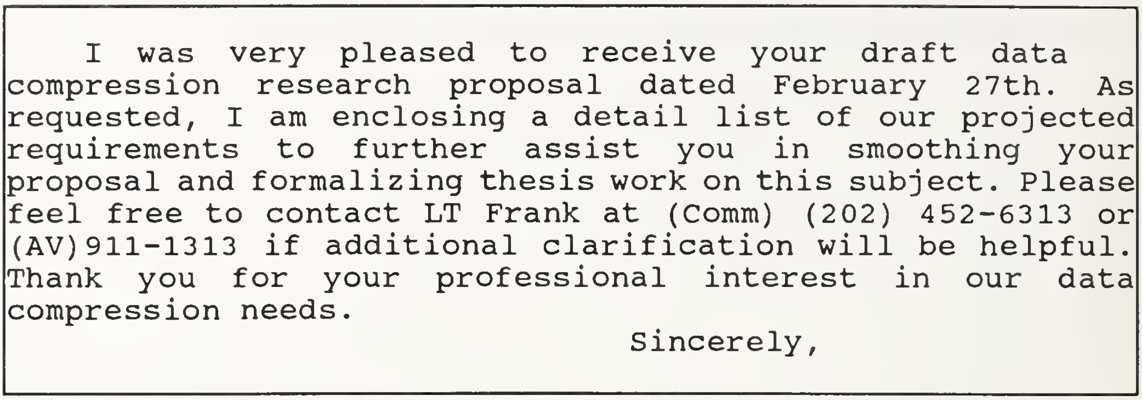
Figure 20. Expansion ratio plot of improved algorithm.

partitions for S_U, S, S_L (Figure 18), to S_A, S', S_B (Figure 19).

D. TRANSLATION EXPERIMENTS

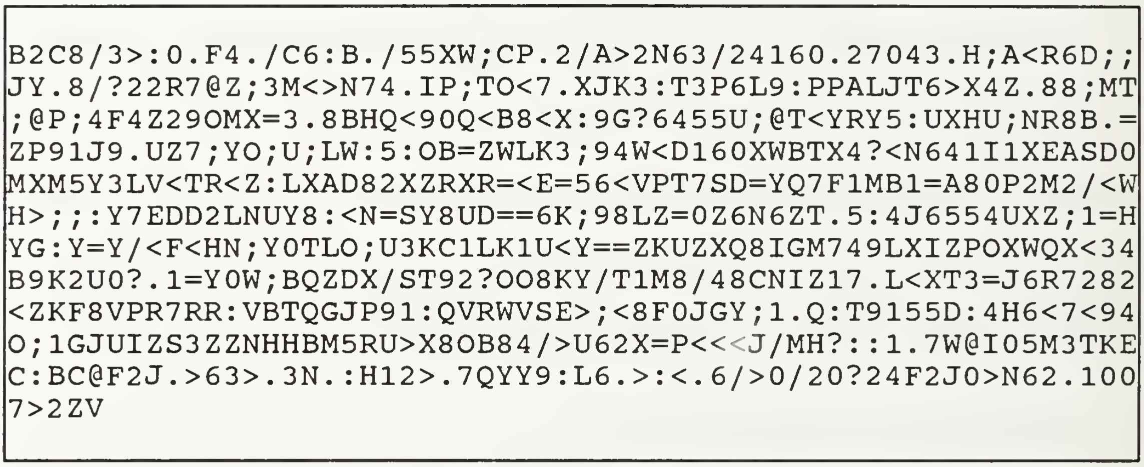
Figure 21 shows a short ASCII file that is to be processed through data compression, data encryption, and character set translation (phone numbers are not real). The compressed and encrypted file after character set translation is shown in Figure 22; all characters appeared within '.' and 'Z' as desired (note that, there is no CR nor LF, the file is displayed in multiple lines for readability). Because the original file is small and therefore is resistant to compression; file size in Figure 22 is larger than that of original file.

From the discussion in Section C.3. (Equation (20)), the theoretical compressed and translated file size can be reduced to : $39.3\% \times (1 + 49.17\%) \simeq 58.62\%$. This



```
   I was very pleased to receive your draft data
compression research proposal dated February 27th. As
requested, I am enclosing a detail list of our projected
requirements to further assist you in smoothing your
proposal and formalizing thesis work on this subject. Please
feel free to contact LT Frank at (Comm) (202) 452-6313 or
(AV)911-1313 if additional clarification will be helpful.
Thank you for your professional interest in our data
compression needs.
                              Sincerely,
```

Figure 21. A sample original text file.



```
B2C8/3>:0.F4./C6:B./55XW;CP.2/A>2N63/24160.27043.H;A<R6D;;
JY.8/?22R7@Z;3M<>N74.IP;TO<7.XJK3:T3P6L9:PPALJT6>X4Z.88;MT
;@P;4F4Z29OMX=3.8BHQ<90Q<B8<X:9G?6455U;@T<YRY5:UXHU;NR8B.=
ZP91J9.UZ7;YO;U;LW:5:OB=ZWLK3;94W<D160XWBTX4?<N641I1XEASD0
MXM5Y3LV<TR<Z:LXAD82XZRXR=<E=56<VPT7SD=YQ7F1MB1=A80P2M2/<W
H>;;:Y7EDD2LNUY8:<N=SY8UD==6K;98LZ=0Z6N6ZT.5:4J6554UXZ;1=H
YG:Y=Y/<F<HN;Y0TLO;U3KC1LK1U<Y==ZKUZXQ8IGM749LXIZPOXWQX<34
B9K2U0?.1=Y0W;BQZDX/ST92?OO8KY/T1M8/48CNIZ17.L<XT3=J6R7282
<ZKF8VPR7RR:VBTQGJP91:QVRWVSE>;<8F0JGY;1.Q:T9155D:4H6<7<94
O;1GJUIZS3ZZNHHBM5RU>X8OB84/>U62X=P<<<J/MH?::1.7W@I05M3TKE
C:BC@F2J.>63>.3N.:H12>.7QYY9:L6.>:<.6/>0/20?24F2J0>N62.100
7>2ZV
```

Figure 22. After compression, encryption and translation.

improved algorithm has been applied to the same set of testing files used in Table IV with three more "PAK" files. The results are listed in Table IV. Expansion ratios in Table II are consistent with expression (19). The PAK files in "C&R" column are reduced to 58.23%, 59.78%, 58.94%, or 54.76% of original file sizes; these are very close to the theoretical value 58.62%.

The discussion in this chapter assumed that the occurrences of all bit patterns are equally likely in the input to the translation algorithm. This is a reasonable assumption since the input bit patterns are dictated by the pointer values in unknown

Table IV. Testing result of improved translation algorithm.

		IMPROVED TRS.		C&R	
FILE	SIZE	SIZE	η	SIZE	%ORG
TXT	24969	37243	49.16%	13853	55.48%
WPR	25195	37570	49.12%	15082	59.86%
CPG	17325	25864	49.29%	6703	38.69%
EXE	24630	36745	49.19%	19695	79.90%
PAK	76644	114625	49.56%	44626	58.23%
PAK1	39024	58327	49.46%	23327	59.78%
PAK2	104622	156516	49.60%	61662	58.94%
PAK3	174226	260362	49.44%	95402	54.76%
IMPROVED TRS. : Improved translation algorithm.					

compression/encryption algorithms. When bit patterns are not equally likely the translation algorithm may be more sophisticated but may achieve a better expansion ratio. For example, if a text file to be processed does not require compression nor encryption the expansion ratio of the character set translation should be smaller than the theoretical 49.17% because the first '0' bit and/or the second '0' bit of each input byte may be skipped in the translation process.

V. CONCLUSIONS

Although the key management scheme discussed in this report is perfectly feasible, it is by no means the only or the best possibility. The method employed here allows only for the availability of different keys for different links and hosts but does not differentiate the different functions or activities for which the keys are used. The functions stressed in this study are data compression and character translation; however, the host system is most likely more versatile. Moreover, the transmissions between hosts, remotes and hosts, remotes and remotes, if independent of each other in encryption, may provide much better protection. These important issues could be solved by a key management scheme based on the popular private-key algorithm, DES. This is beyond the scope of this thesis. Nevertheless, its possibilities introduce a worthwhile follow-on research.

Finally, to make all algorithms and source code completely transportable among hardware/operating system environments, the work of error detection and correction becomes an absolute necessity. It is possible to use 'checksum' or CRC techniques to detect transmission errors by attaching d characters to each block of b characters. These d checking characters (D) are computed from the b information characters (B). Traditional checksum is not capable of locating which byte in B is in error since characters in B may have $b!$ permutations and some permutations may result in the same D. To facilitate the error correction we may have to use some non-commutative operations in the construction of D from B. For instance, characters in B are arranged in two matrices and their product is used as D. Because matrix multiplication is non-commutative it may be a starting point for character-oriented error detection and correction. Other powerful error correction codes such as R-S (Reed-Solomon codes) are also available for further study.

APPENDIX A. PROGRAM LISTINGS

```
/***********************************************************

   This is a routine program to determine whether the input
   file is compressed file, compressed and encrypted file,
   compressed and translated file, or mixed all of three;
   the routine will recover, decrypt, or uncompress
   according to the command line options which given by
   main().
                                              Tsai Chien-C

***********************************************************/

/******************************

  Header files and local
  variables.

******************************/


#include "zip.h"
#include <math.h>

#define KEYLEN 12           /* included '\0' for sure     */
#define MASK 0377           /* capture the lower 8 bits   */
#define ROTORSZ 256         /* limited within byte pattern*/
#define BASEMOD 100000000L  /* modular number for password
                               conversion                 */
#define MODNUMBER 2147483641L
                            /* 2^31, mod number, good for
                               'long' operation, it can be
                               expanded to increase the
                               security of password if
                               needed                     */
#define TF1 "zzzzzzzz.111"  /* temporary translated file  */
#define TF2 "zzzzzzzz.222"  /* temporary recovered file   */
#define TF2 "zzzzzzzz.333"  /* temporary (de)encrypted
                               file                       */
#define INF "oooooooo.000"  /* temporary input file if any
                               operation is performed     */
local long SERVKEY;         /* modulated service key number
                                                          */
```

```
local long MASTERKEY;       /* modulated master key number*/
local long TEMPKEY;
local long double ANSWERKEY;/* master key number after
                               computation                */
local char pass_word[KEYLEN];
                            /* password from stdin or
                               compressed file header     */
local char code1[ROTORSZ];  /* three (de)encrypt index
                               random code tables         */
local char code2[ROTORSZ];
local char code3[ROTORSZ];
local char deck_num[ROTORSZ];
                            /* for use with shuffle()
                               presented                  */
local char keyword_buf[11]; /* get password as the key to
                               generate random table      */
local char *translated = "B2C\0";
                            /* the first three bytes of
                               translated file            */
local char *encrypted = "!2?\0";
                            /* the first three bytes of
                               encrypted file             */
FILE *tempf1, *tempf2, *tempf3;
                            /* temporary files generated by
                               trans(), recov(), and
                               encrypt()                  */
FILE *infile;               /* input file from main()     */

/***********************************************************

   Function first_pass() get the address of 'zipfile' from
   main() pass to verify() for file type operation, a output
   file is to be renamed according to the parameter produced
   by verify(), then return the parameter with a original
   'zipfile' file name but point to new address.

***********************************************************/

int first_pass()
{
  int ftype;
  char *original;

  pass_word[11] = '\0';     /* take only 11 bytes password*/
  strcpy( original, zipfile );
                            /* keep the input file name   */
  objectfile = zipfile;     /* get the input file address */
  ftype = verify();         /* pass to file type operations
                                                          */
  if ( ftype != 1 && ftype != 5 )
```

```
     rename( zipfile, INF );
                            /* file had been recovered or
                               decrypted, keep original
                               input file in case of any
                               error during decompress
                               operation; otherwise unlink
                               it                         */
     switch ( ftype )
        {
         case 1:  case 5:
            break;          /* no operation is performed,
                               either a null file ( output
                               file name after compressed )
                               or purely compressed file */
         case 2:
            rename( TF3, original );
            strcpy( zipfile, original );
            break;          /* file was decrypted, now
                               change the file name to the
                               same as input file, but new
                               address                    */
         case 3:
            rename( TF2, original );
            strcpy( zipfile, original );
            break;          /* file was recovered,substitute
                               with original name but new
                               address                    */
         case 4:
            rename( TF3, original );
            strcpy( zipfile, original );
            unlink( TF2 );  /* file was recovered and
                               decrypted, delete temporary
                               file after file name switched
                                                          */
        }
     printf( "%s\n", zipfile );
     objectfile = zipfile;  /* to be used later in trans()
                               and encrypt()              */
     return ftype;          /* back to main(), 'ftype'
                               indicate what operation ever
                               been performed             */
}

/***********************************************************

    Function second_pass() determine whether a compressed
    output file to characters translation, encryption or not
    by remember the command line options or the parameters
    in first_pass(). then assign a desired output file name
    and delete all temporary files.
```

```
************************************************************/

second_pass()
{
 if ( to_encrypt == 1 ) encrypt( to_encrypt );
                            /* go encryption if command line
                               specified or input file it
                               was                          */
 if ( to_trans == 1 ) trans();
                            /* go characters translation if
                               command line specified or
                               input file it was            */
 if ( to_trans || to_encrypt ) unlink( zipfile );
                            /* now delete the temporary file
                               if either trans() or
                               encrypt() was taken          */
 unlink( INF );             /* the original input file now
                               is useless                   */
 if ( to_trans && to_encrypt || to_trans && !to_encrypt )
     rename( TF1, zipfile );/* once trans() was performed,
                               the output file will be TF1*/
 else if ( !to_trans && to_encrypt ) rename( TF3, zipfile );
                            /* encryption only, then output
                               TF3                          */
 else return;               /* nothing happen, go back to
                               main()                       */
 exit( 0 );                 /* otherwise, quit here         */
}

/************************************************************

  The function verify() is a main routine for each
  operation, it opens the input file, check the header and
  determine what kind operation should be performed then
  pass a parameter back to function first_pass().

************************************************************/

int verify()
{
 char identify[3];
 int ftype, i;

 if (( infile = fopen( objectfile, "rb" )) == NULL )
    return ftype = 1;       /* just a output file name
                               assigned by user             */
 else
    {
     for ( i = 0; i <= 2; i++ )
        identify[i] = fgetc( infile );
```

```
                       /* get the header             */
if ( strncmp( identify, translated, 3 ) == 0 )
   {
                       /* input file was translated  */
    to_trans = 1;      /* set index, file should back
                          to same form after operations
                          in main()                  */
    recov();           /* recover from a translated
                          file                       */
    fclose( infile );
    infile = fopen( TF2, "rb" );
                       /* open the temporary file
                          generated by recov()       */
    for ( i = 0; i <= 2; i++ )
       identify[i] = fgetc( infile );
                       /* get the header again       */
    if ( strncmp( identify, encrypted, 3 ) == 0)
       {
                       /* input file also was encrypted
                                                     */
        ftype = 4;
        for ( i = 0; i < 11; i++ )
           pass_word[i] = fgetc( infile ) + 40;
                       /* now get the encrypted
                          password header and convert
                          to real keyword            */
        to_encrypt = process_passw();
                       /* go check if the user's
                          password is matched        */
        if ( to_encrypt == 1 ) encrypt( ftype );
                       /* verified! and decrypt it   */
        coded = 1;     /* remember it, random table
                          can't random again         */
        return ftype;  /* it was a compressed,
                          encrypted and translated file
                                                     */

       }
    else
       {
        fclose( infile );
        return ftype = 3;
                       /* it was a compressed and
                          translated file            */
       }
   }
else if ( strncmp( identify, encrypted, 3 ) == 0 )
   {
    ftype = 2;         /* it was a compressed and
                          encrypted file             */
    for ( i = 0; i < 11; i++ )
```

```
            pass_word[i] = fgetc( infile ) + 40;
                            /* get the encrypted password
                               header and convert to real
                               keyword                    */
         to_encrypt = process_passw();
                            /* go check if the user's
                               password is matched        */
         if ( to_encrypt == 1 ) encrypt( ftype );
                            /* verified! and decrypt it   */
         coded = 1;
         fclose( infile );
         return ftype;
        }
     else
        {
         fclose( infile );
         return ftype = 5;  /* it was just a compressed file
                               but not in translated or
                               encrypted form             */
        }
    }
}

/***********************************************************

   The function process_passw() verify the user's password
   by implemented the "Masterkey" scheme, both user password
   and service ( compressed package ) password are decrypt
   to a 10 digit ( long ) number by function passw2num()
   first, the 10 digit number should not larger than mod
   number which defined as 2147483641; further, the user's
   number is computed according to their list of service
   key, again mod by 2147483641 to compare with services
   number, return '1' if equal, otherwise let user has
   another try. Primes number and service list in this
   function can be modified to allow system expansion and
   become flexible. Mod number can be changed to a larger
   number too, which will increase the password complexity.
   In this experiment, I built a 70 services system, and 8
   masterkey user, details was told in Chapter "Multilevel
   security".

***********************************************************/

int process_passw()
{
 int try = 1, i, j, k, s_path;
 char passw_in[KEYLEN];
 int P[70] =
         {409, 401, 397, 389, 383, 379, 373, 367, 359, 353,
             349, 347, 337, 331, 317, 313, 311, 307, 293, 283,
             281, 277, 271, 269, 263, 257, 251, 241, 239, 233,
             229, 227, 223, 211, 199, 197, 193, 191, 181, 179,
             173, 167, 163, 157, 151, 149, 139, 137, 131, 127,
             113, 109, 107, 103, 101,  97,  89,  83,  79,  73,
              71,  67,  61,  59,  53,  47,  43,  41,  37,  31};
                               /* 70 prime numbers from 31,
                                  each assigned to a service */
    int s_list[8][70] =
                   {{ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
                     10, 11, 12, 13, 14, 15, 16, 17, 18,
                     20, 21, 22, 23, 24, 25, 26, 27,
                     30, 31, 32, 33, 34, 35, 36,
                     40, 41, 42, 43, 44, 45,
                     50, 51, 52, 53, 54,
                     60, 61, 62, 63, -1                     },
                               /* services list of masterkey
                                  #1, the numbers matched the
                                  primes in P[]              */
                    { 0,  1,  2,  3,  4,  5,  6,
                     10, 11, 12, 13, 14, 15, 16, -1         },
                               /* services list of masterkey #2
                                                             */
                    {40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
                     -1                                     },
                               /* services list of masterkey #3
                                                             */
                    {20,
                     30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
                     -1                                     },
                               /* services list of masterkey #4
                                                             */
                    { 0,  1,  2,  3,
                     10, 11, 12, 13,
                     20, 21, 22, 23,
                     30, 31, 32, 33,
                     40, 41, 42, 43,
                     50, 51, 52, 53,
                     60, 61, 62, 63, -1                     },
                               /* services list of masterkey #5
                                                             */
                    { 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
                     10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                     20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
                     30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
                     40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
                     50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
                     60, 61, 62, 63, 64, 65, 66, 67, 68, -1 },
                               /* services list of masterkey #6
                                                             */
```

```
                 { 0,
                  10,
                  20,
                  30,
                  40,
                  50,
                  60, -1                                 },
                            /* services list of masterkey #7
                                                          */
                 {40, 41,
                  50, 51, -1                             }};
                            /* services list of masterkey #8
                                                          */

      passw_in[11] = '\0';  /* make sure no garbage follows
                                                          */
      printf( "File was encrypted, please...\n" );
loop: printf( "\nEnter password: " );
                            /* request for user's password*/
      scanf( "%s", passw_in );
      if ( strncmp( passw_in, pass_word, 11 ) == 0 )
         return 1;
                            /* the password is exactly the
                               same as that got from file
                               header, it indicated the user
                               was the one who encrypted it,
                               of course he is authority to
                               access                     */
      if ( strlen( passw_in ) != 11 )
         {
          printf( "Invalid password! " );
                            /* we don't consider a password
                               other than 11 characters, so
                               give him one more try      */
          if ( try == 3 )
             {
              printf( "Sorry, no lucky guess, good bye!" );
              if ( to_trans == 1 ) unlink( TF2 );
              exit( 0 );    /* triple wrong guesses, who is
                               the boss? bye anyway. check
                               if file was recovered, then
                               delete the temporary file
                               generated by recov()       */
             }
          ++try;
          goto loop;        /* try one more if not reach
                               three                      */
          }
 k = passw2num( passw_in, 1 );
                            /* take care the user's password
```

```
                              first convert it to 10 digit
                              number                     */
s_path = passw2num( pass_word, 0 );
                           /* then convert the password in
                              file header                */
s_path = 80 - s_path;      /* determine where the service
                              located                    */
ANSWERKEY = ( long double ) MASTERKEY;
                           /* initialize the base number,
                              it is, in fact, the masterkey
                              number                     */
TEMPKEY = MASTERKEY;
printf( "Verifying" );
j = 0;
if ( k <= 7 )              /* it is a masterkey user     */
   for ( i = 0; i <= 69; i++ )
      {
       if ( i == s_list[k][j] )
          {
                           /* care only the one matched
                              with services list         */
           if ( i < ( s_path - fmod( s_path, 10 ))
                                   || i > s_path )
              {
                           /* the overlay parts ( with
                              service key ) will not be
                              considered                 */
               keynum_mod( P[i] );
                           /* now we got what we want,
                              let's roll it              */
               TEMPKEY = ( long ) ANSWERKEY;
              }
           ++j;            /* update service list to next
                              one                        */
          }
       if ( i % 5 == 0 ) printf( "." );
                           /* just tell you I am working */
      }
else
   {
    k = 80 - k;            /* it is another service key, we
                              have to find out the
                              relationship in between, see
                              if it > the encrypted one  */
    for ( i = 0; i <= 69; i++ )
       {
        if ( i > s_path && i <= k )
          {
                           /* consider those prime numbers
                              between two services only  */
```

```
            keynum_mod( P[i] );
            TEMPKEY = ( long ) ANSWERKEY;
           }
         if ( i % 5 == 0 ) printf( "." );
        }
     }
 printf( "\n" );
 if ( SERVKEY == ( long ) ANSWERKEY ) return 1;
                            /* return greeting signal if
                               user's number matched      */
 else
    {                       /* else please try again if not
                               over three times yet       */
     printf( "Invalid password!" );
     if ( try == 3 )
        {
         printf( " Sorry, no lucky guess, good bye!" );
         if ( to_trans == 1 ) unlink( TF2 );
         exit( 0 );
        }
     ++try;
     goto loop;
    }
}

/***********************************************************

   The function passw2num() convert each password to a 10
   digit ( long ) number by a random pattern which assigned
   by programmer. then return a value as service list ( or
   service number ) to be referenced.

***********************************************************/

int passw2num( pass_2_num, mkorsk )
char pass_2_num[KEYLEN];
int mkorsk;
{
 int k, head, i=1;
 char ch_to_convert[KEYLEN];

 pass_2_num[11] = '\0';     /* make sure no garbage follow*/
 ch_to_convert[11] = '\0';  /* each character will convert
                               to 0 - 9 ASCII and put into
                               here                       */
 head = pass_2_num[0] + 3;  /* take care first character
                               with + 100 - 97, '+ 100' is
                               to avoid a situation of
                               number start with 0        */
 itoa( head, ch_to_convert, 10 );
```

```
                            /* put first two digit into
                               buffer                     */
 ch_to_convert[3] = pass_2_num[1] - 49;
 ch_to_convert[4] = pass_2_num[2] - 52;
 ch_to_convert[5] = pass_2_num[3] - 55;
                            /* convert 2nd, 3rd, 4th
                               character to be 3rd, 4th, 5th
                               digit in buffer            */
 ch_to_convert[6] = pass_2_num[4];
 ch_to_convert[7] = pass_2_num[5];
                            /* just copy the 5th and 6th
                               character to be 6th and 7th
                               digit                      */
 ch_to_convert[8] = pass_2_num[6] - 58;
 ch_to_convert[9] = pass_2_num[7] - 61;
 ch_to_convert[10] = pass_2_num[8] - 64;
                            /* convert 7th, 8th and 9th
                               character to be 8th, 9th, and
                               10th digit in buffer       */
 k = atoi( pass_2_num + 9 );/* 10th and 11th character
                               referenced as the service
                               list or service number     */
 while( ch_to_convert[i] == '0' ) ++i;
                            /* before convert to a ( long )
                               type, let's skip those
                               useless '0'                */
 if ( mkorsk ) MASTERKEY = atol( ch_to_convert + i );
 else SERVKEY = atol( ch_to_convert + i );
                            /* convert to ( long ) type
                               begin with non-zero digit  */
 return k;
}

/***********************************************************

   The function keynum_mod() calculate the result of mod(
   MASTERKEY*MASTERKEY ) up to 'p' times loops, 'p' is the
   prime number of each service. The computation period
   could be longer if a service list has more members, but
   the result will not over mod number 2147483641 since it
   keep mod operation on each result.

***********************************************************/

keynum_mod( p )
int p;
{
    int i;

    for ( i = 2; i <= p; i++ )
```

```
        {
         ANSWERKEY *= ( long double ) TEMPKEY;
         if ( ANSWERKEY >= MODNUMBER ) ANSWERKEY = fmodl(
              ANSWERKEY, MODNUMBER );
         }
}

/************************************************************

   The function encrypt() is part of the "crypt.c" routine
   in UNIX which written by Berkeley 1985; It is a one-rotor
   machine designed along the lines of Enigma but
   considerably trivialized, the key to generate a random
   table for encryption is just the password from user who
   request encryption, but is the header from input file
   when decrypting.

************************************************************/

encrypt( enc )
int enc;
{
 register i, n1, n2, nr1, nr2;
 char code[KEYLEN];
 int try = 1;

 code[11] = '\0';           /* this is keyword buffer to
                               generate random table, make
                               sure no garbage follow     */
 if ( enc == 1 && !coded )
    {
                            /* only if user is to encrypt
                               the compressed file        */
lop: printf( "Enter encrypted password( SERVKEY ): " );
     scanf( "%s", pass_word );
     if ( strlen( pass_word ) != 11 ¦¦ *( pass_word + 4 ) >
        57 ¦¦ *( pass_word + 5 ) > 57 ¦¦ *( pass_word + 9 )
        > 57 ¦¦ *( pass_word + 10 ) > 57 )
        {
                            /* check the service ( user )
                               password carefully to avoid
                               a encrypted file become
                               unaccessible               */
         printf( "Invalid password!\n" );
         if ( try == 3 )
            {
             printf( " Sorry, can't encrypt with password\
                       .\n" );
             exit( 0 );     /* left file as it was (
                               compressed ) after three
```

```
                              tries                      */
           }
         ++try;
         goto lop;
       }
    printf( "Encrypted!...\n" );
   }
tempf3 = fopen( TF3, "wb" );
                           /* open temporary file for
                              (de)encryption output      */
if ( enc == 1 )            /* came from second_pass(), do
                              encryption only            */
   {
    infile = fopen( objectfile, "rb" );
                           /* the input file was compressed
                                                         */
    fwrite( encrypted, 1, 3, tempf3 );
    for ( i = 0; i < 11; i++ )
       fputc( pass_word[i] - 40, tempf3 );
                           /* write 3 byte header and
                              shifted password into output
                              file                       */
   }
else printf( "Decrypted!...\n" );
                           /* came from first_pass()     */
strncpy( code, pass_word, 11 );
                           /* password is the key to
                              generate random code table */
if ( !coded )
   rand_code( code );      /* random table can't be random
                              again if decryption and
                              encryption performs in same
                              target file                */
else printf( "Encrypted!...\n" );
n1 = 0;
n2 = 0;
nr2 = 0;                   /* initial index number       */
while (( i = fgetc( infile )) != EOF )
   {
    nr1 = n1;
    i = code2[( code3[( code1[( i + nr1 ) & MASK] + nr2 )
                          & MASK] - nr2 ) & MASK] - nr1;
                           /* shift input character alone
                              the random code table      */
    fputc( i, tempf3 );    /* then output encrypted byte */
    n1++;
    if ( n1 == ROTORSZ )   /* keep index number in the
                              range of 256               */
       {
        n1 = 0;
```

```
         n2++;
         if ( n2 == ROTORSZ ) n2 = 0;
         nr2 = n2;
        }
    }
 fclose( tempf3 );
 fclose( infile );
}

/************************************************************

   The function rand_code() generate the random code table
   according to the key ( password ) from encrypt().

************************************************************/

rand_code( pw )
char pw[KEYLEN];
{
 int ic, i, k, t;
 unsigned random;
 long seed;

 strncpy( keyword_buf, pw, 11 );
                              /* get the key                 */
 for ( i = 0; i < 11; i++ )
    pw[i] = '\0';             /* clear the keyword buffer
                                 exactly 11 bytes, go any
                                 further will ruin the
                                 consecutive memory location
                                 and destroy the performance*/
 seed = 123;
 for ( i = 0; i < 11; i++ )
    seed = seed * keyword_buf[i] + i;
                              /* growing seed number with key
                                                             */
 for ( i = 0; i < ROTORSZ; i++ )
    {
     code1[i] = i;            /* initial the index table     */
     deck_num[i] = i;         /* not use in this routine     */
    }
 for ( i = 0; i < ROTORSZ; i++ )
    {
     seed = 5 * seed + keyword_buf[i % 11];
                              /* rolling key buffer to produce
                                 different seed number each
                                 time                        */
     random = seed % 65521;   /* random should not over 16 bit
                                                             */
     k = ROTORSZ - 1 - i;     /* decrease table index as the
```

```
                               loop growing               */
     ic = ( random & MASK ) % ( k + 1 );
                            /* offset index decreasing as
                               index decreasing           */
     random >>= 8;
     t = code1[k];
     code1[k] = code1[ic];
     code1[ic] = t;         /* swap index table           */
     if ( code3[k] != 0 ) continue;
                            /* go generate next code if
                               buffer is not null         */
     ic = ( random & MASK ) % k;
     while ( code3[ic] != 0 ) ic = ( ic + 1 ) % k;
     code3[k] = ic;
     code3[ic] = k;         /* if present buffer location is
                               zero then we need a value
                               fill it by scan and swap the
                               index                      */
    }
 for ( i = 0; i < ROTORSZ; i++ ) code2[code1[i] & MASK] = i;
                            /* generate second random table
                               by first table as index    */
}

/***********************************************************

   The function trans() check every 6-bit pattern from input
   file, if the decimal value larger than 31 but less than
   45, then translate this 6-bit to a character between 'N'
   and 'Z', otherwise shift 1 bit left and translate a
   5-bit pattern to a character between '.' and 'M', the
   shifted bit then become the MSB of next 6-bit pattern.

***********************************************************/

trans()
{
 char p = 'A' & 0x00;
 char *outfile, *b, c;
 int i, bo = 0, bs = 0, last = 1, n = 9;
 unsigned char r[9] =
   { 0x00, 0x01, 0x03, 0x07, 0x0f, 0x1f, 0x3f, 0x7f, 0xff };
 unsigned char l[9] =
   { 0x00, 0x80, 0xc0, 0xe0, 0xf0, 0xf8, 0xfc, 0xfe, 0xff };
 float out_bytes = 0, in_bytes = 0;

 tempf1 = fopen( TF1, "w" );
 if ( !to_encrypt )
    tempf3 = fopen( objectfile, "rb" );
                            /* no encrypted, no TF3
```

```
                               temporary, the input file
                               will be the compressed one */
 else tempf3 = fopen( TF3, "rb" );
                            /* otherwise, get it from
                               encrypt()                  */
 for ( i = 0; i <= 2; i++ )
    fputc( *translated++, tempf1 );
                            /* write translated file header
                                                          */
 out_bytes = 3;
 for ( ; ; )
    {
     i = 0;
     while ( !feof( tempf3 ))
        {
         *b++ = fgetc( tempf3 );
         ++i;
         if ( i == n ) break;
                            /* input 9 bytes each time, good
                                for LZW too               */
        }
     if ( i < n )
        {
         last = 0;
         n = i;             /* here comes a EOF, just
                               remember how many byte left
                               in buffer                  */
        }
     in_bytes += n;         /* input byte count           */
     b -= n;                /* back to start address      */
     for ( i = 1; i <= n; i++ )
        {
same:    if ((((( *b & l[6 - bs] ) >> ( 2 + bs )) ¦ p ) >=
            32 ) && (((( *b & l[6 - bs] ) >> ( 2 + bs )) ¦
             p ) < 45 ))
            {
                            /* get 6-bit pattern each time,
                               see if the decimal value
                               between 32 and 44          */
             fputc((( *b & l[6 - bs] ) >> ( 2 + bs )) ¦ p )
                   + 46, tempf1 );
                            /* yes, then translate the 6-bit
                               to the character between 'N'
                               and 'Z'                    */
             ++out_bytes;   /* update output byte count   */
             bo += 6;       /* update output bit count    */
            }
         else
            {
             fputc((( *b & l[5 - bs] ) >> ( 3 + bs )) ¦
                 ( p >> 1)) + 46, tempf1 );
                           /* no, then shift 1 bit right
                              and translate the 5-bit to
                              the character between '.'
                              and 'M'                    */
            ++out_bytes;

            bo += 5;
           }
        bs = 8 * i - bo;   /* remember how many bit left */
        if ( bs >= 6 )
           {
            p = ( *b & r[bs] ) >> ( bs - 6 );
                           /* if left bits is more than 5,
                              no input from buffer is
                              necessary, now composite the
                              next 6-bit pattern         */
            bs = 6;
            goto same;     /* go check again             */
           }
        else
           {
            p = ( *b & r[bs] ) << ( 6 - bs );
            ++b;           /* otherwise shift left bits to
                              MSB and update input buffer,
                              get next byte again        */
           }
       }
    if ( last == 0 )       /* the EOF case               */
       {
        if (( p >= 32 ) && ( p < 45 ))
           fputc( p + 46, tempf1 );
        else fputc(( p >> 1 ) + 46, tempf1 );
                           /* output the last bit pattern
                              by followed the same rule  */
        ++out_bytes;
        unlink( TF3 );     /* temporary output file of
                              encrypt() is useless now   */
        fclose( tempf1 );
        break;             /* no more translate is needed*/
       }
    b -= n;                /* keep working, back to start
                              address ready for next 9-byte
                              from input file            */
    bo = 0 - bs;           /* output bit is bit needed for
                              next 6-bit pattern anyway  */
   }
printf( "Translated! expansion rate : %5.2f%%...\n",
        ( out_bytes - in_bytes ) * 100 / in_bytes );
                           /* expanded ratio is the exceed
```

```
                               byte percentage of inputfile
                                                          */
}

/************************************************************

   The function recov() get byte one by one from input file,
   check it if fall between 'N' and 'Z' then concatenate the
   6 LSB to the output buffer, otherwise concatenate the 5
   LSB to the output buffer.

************************************************************/

recov()
{
 unsigned char left[9] =
   { 0x00, 0x80, 0xc0, 0xe0, 0xf0, 0xf8, 0xfc, 0xfe, 0xff };
 int size, bi, i, b_i = 0, bs = 0, n = 9;
 char *b, p = 'A' & 0x00;

 tempf2 = fopen( TF2, "wb" );
 for ( ;; )
    {
     size = 0;
     *b = p;                /* output buffer fill with the
                               bits last loop left        */
     b_i = bs;
     if (( p = fgetc( infile )) == EOF ) break;
                            /* quit if end of file, go take
                               care output buffer         */
     p -= 46;               /* shift back to original value
                                                          */
     if ( p < 32 ) bs = 5;  /* if the decimal value less
                               than 32, we have 5 bits to be
                               plug into output buffer    */
     else bs = 6;           /* otherwise concatenate 6 bits
                                                          */
lop: ++size;
     while ( size <= n )    /* up to 9 bytes in output
                               buffer                     */
        {
         while ( bs > 0 )   /* fill into buffer if any bit
                               available                  */
            {
             *b = *b & left[b_i] ¦ (( p <<
                                       ( 8 - bs )) >> b_i );
                            /* concatenate output buffer to
                               a byte pattern             */
             bi = min( bs, 8 - b_i );
                            /* the inserted bits should not
```

```
                               exceed either the available
                               bits or the bits buffer
                               needed                     */
             bs -= bi;      /* update the bits available  */
             b_i += bi;     /* update the inserted bits   */
             if ( b_i >= 8 )
                {
                 ++b;       /* update output buffer if bits
                               inserted more than 7       */
                 b_i = 0;   /* clear inserted bit count   */
                 goto lop;
                }
            }
         if (( p = fgetc( infile )) == EOF ) break;
                            /* input another character    */
         p -= 46;
         if ( p < 32 ) bs = 5;
         else bs = 6;
        }
     b -= ( size - 1 );     /* buffer is full, back to start
                               address, right time for
                               output                     */
     if (( size - 1 ) != n )/* case of EOF met somewhere in
                               between                    */
        {
         for ( i = 1; i < ( size - 1 ); i++ )
            fputc( *b++, tempf2 );
         break;             /* output whatever have and quit
                                                          */
        }
     p <<= ( 8 - bs );      /* keep going, then shift last
                               character to right position*/
     for ( i = 1; i <= ( size - 1 ); i++ )
        fputc( *b++, tempf2 );
     b -= n;                /* not EOF yet, output 9 bytes
                               and back to start address
                               ready for next loop        */
    }
 fclose( tempf2 );
 printf( "Recovered!...\n" );
}
```

APPENDIX B. MULTILEVEL EXPERIMENT REFERENCE TABLES

Table A. Prime Numbers Distribution.

	0	1	2	3	4	5	6
9	353	283	233	179	127	73	31
8	359	293	239	181	131	79	37
7	367	307	241	191	137	83	41
6	373	311	251	193	139	89	43
5	379	313	257	197	149	97	47
4	383	317	263	199	151	101	53
3	389	331	269	211	157	103	59
2	397	337	271	223	163	107	61
1	401	347	277	227	167	109	67
0	409	349	281	229	173	113	71
e.g. S_{26}, Prime = 251.							

Table B. Service Key Numbers and Passwords.

SK	PRIMES	KEY NUMBERS	CRYPTED PASSWORDS
00	409	0015206469	abii06nsy80
01	401	0205124689	caih24puy79
02	397	1440799106	oedn99kmv78
03	389	0698488394	gjlk88mvt77
04	383	0850186419	ifdh86nny76
05	379	0523243868	fcgi43rsx75
06	373	0150800082	bfdo00jur74
07	367	1537810080	pdko10jup73

08	359	0883797530	iign97opp72
09	353	2138861890	vdlo61rvp71
10	349	0941660305	jeem60mmu70
11	347	1425676110	ocim76knp69
12	337	0098817834	ajlo17rpt68
13	331	0947247270	jeki47ltp67
14	317	0699935923	gjmp35sos66
15	313	1262712566	mgfn12osv65
16	311	1600459268	qadk59lsx64
17	307	0953599917	jfgl99snw63
18	293	2082398675	uifj98ptu62
19	283	0072782989	ahfn82suy61
20	281	0112561142	bbfl61kqr60
21	277	1243547112	megl47knr59
22	271	0442572687	eefl72puw58
23	269	1039735727	kdmn35qow57
24	263	0907973227	jakp73low56
25	257	1170134111	lhdh34knq55
26	251	1545019722	peig19qor54
27	241	1347280362	neki80msr53
28	239	2119781043	vbmn81jqs52
29	233	1547062091	pekg62jvq51
30	229	1775718672	rhin18ptr50
31	227	1475717646	ohin17pqv49
32	223	0985717941	jiin17sqq48
33	211	1760752331	rgdn52mpq47
34	199	0642865897	gefo65rvw46
35	197	1989952527	timp52oow45
36	193	0504668756	fahm68qrv44
37	191	0030306011	addj06jnq43

38	181	1874146998	shhh46svx42
39	179	0427977764	eckp77qst41
40	173	0491586618	ejel86pnx40
41	167	0866484154	igjk84krt39
42	163	0265736240	cgin36lqp38
43	157	0256917344	cfjp17mqt37
44	151	0516007035	fbjg07jpu36
45	149	1636320334	qdjj20mpt35
46	139	0524932628	fchp32pox34
47	137	1654493248	qfhk93lqx33
48	131	0374636578	dhhm36otx32
49	127	1650980892	qfdp80rvr31
50	113	1468654363	oglm54mss30
51	109	0824711795	ichn11qvu29
52	107	1147133098	lekh33jvx28
53	103	0936296552	jdji96orr27
54	101	0813794276	ibgn94ltv26
55	097	2066215471	ugji15ntq25
56	089	0266019527	cgjg19oow24
57	083	0880884223	iido84los23
58	079	1236583491	mdjl83nvq22
59	073	0842991221	iefp91loq21
60	071	0109676383	bamm76mus20
61	067	1407243086	oaki43juv19
62	061	0688602219	gilm02lny18
63	059	1005217527	kaii17oow17
64	053	0029035185	acmg35kuu16
65	047	1006677351	kajm77mrq15
66	043	1947531213	tekl31lns14
67	041	0487399200	eikj99lmp13

68	037	0179179038	bhmh79jpx12
69	031	0535866065	fdio66jsu11

SK: Services Key.
KEY NUMBER: After mod() operation.
CRYPTED PASSWORDS: Encrypted from Key numbers.

Table C. Master Key Lists

MK	NUMBER	PASSWORD (encrypted)	SERVICE KEY LIST (Each number is an index correspond to each service key in table A)
0	1865519134	sgil19kpt00	09 08 18 07 17 27 06 16 26 36 05 15 25 35 45 04 14 24 34 44 54 03 13 23 33 43 53 63 02 12 22 32 42 52 62 01 11 21 31 41 51 61 00 10 20 30 40 50 60
1	0233536330	cdgl36mpp01	06 16 05 15 04 14 03 13 02 12 01 11 00 10
2	1650980892	qfdp80rvr02	49 48 47 46 45 44 43 42 41 40

3	0002255128	aafi55kox03				39			
						38			
						37			
						36			
						35			
						34			
						33			
						32			
						31			
					20	30			
4	1609630558	qamm30orx04	03	13	23	33	43	53	63
			02	12	22	32	42	52	62
			01	11	21	31	41	51	61
			00	10	20	30	40	50	60
5	1582313220	pifj13lop05	09	19	29	39	49	59	
			08	18	28	38	48	58	68
			07	17	27	37	47	57	67
			06	16	26	36	46	56	66
			05	15	25	35	45	55	65
			04	14	24	34	44	54	64
			03	13	23	33	43	53	63
			02	12	22	32	42	52	62
			01	11	21	31	41	51	61
			00	10	20	30	40	50	60
6	0046844145	aejo44kqu06	00	10	20	30	40	50	60
7	1479772666	ohmn72psv07					41	51	
							40	50	

LIST OF REFERENCES

1. J. McLean, "The Specification and Modeling of Computer Security," *IEEE Computer*, pp. 9-16, Jan. 1990.

2. G. C. Chick and S. E. Tavares, "Flexible Access Control with Master Keys," *Crypto 89, Springer-Verlag,* pp.316-322.

3. FIPS, "Data Encryption Standard," *Federal Information Processing Standards Publication,* 46-1, NBS, Jan. 1988.

4. C. T. Bell, "Better OPM/L Text Compression," *IEEE Transactions on Communication,* vol. COM-34, no. 12, pp. 1176-1182, Dec. 1986.

5. L. J. Bentley, D. D. Sleator, E. R. Tarjan and K. V. Wei, "A Locally Adaptive Data Compression Scheme," *CACM*, pp. Apr. 1986.

6. Ziv I. and Lempel A., "Compression of Individual Sequences Via Variable-Rate Coding," *IEEE Transaction on Information Theory,* Sep. 1978.

7. P. E. Bender and J. K. Wolf, "New Asymptotic Bounds and Improvements on the Lempel-Ziv Data Compression Algorithm," *IEEE Transaction on Information Theory,* vol. 37, no. 3, pp 721-729, May. 1991.

8. T. A. Welch, "A Technique for High Performance Data Compression," *IEEE Comput.*, vol. 17, pp. 8-19, Jun. 1984.

9. M. Adler, R. B. Wales and J-loup Gailly, "ZIP.C," 1991.

10. J. A. Storer and T. G. Szymanski, "Data Compression Via Textual Substitution,"

J. ACM, vol. 29, no. 4, pp 928-951, 1982.

11. PAKWARE, "PKZIP, appnote.txt," Aug. 1991.

12. Jung, "The Performance Comparison of Data Compression Software", *Thesis research, Naval Postgraduate School.* Mar. 1992.

13. S. G. Akl and P. D. Taylor, "Cryptographic Solution to A Multilevel Security Problem," *Crypto 82, Springer-Verlag,* pp. 237-249.

14. R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *CACM,* pp. 120-126, Feb. 1978.

15. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE std* 754-1985.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

2. Library Code 52 2
Naval Postgraduate School
Monterey, CA 93943-5000

3. Chairman, Code EC 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5000

4. Naval Security Group Detachment 1
Corry Station
Pensacola, FL 32511-5100

5. Professor Chyan Yang, Code EC/Ya 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5000

6. Professor Paul H. Moose, Code EC/Me 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, CA 93943-5000

7. MICA 1
P.O. Box 90049 Hsing-Tien
Taipei county 231, Taiwan R.O.C.

8. Jung, Young Je 1
155-17, 12-Tong 3-Ban, Jangjuni-Dong, Kumjong-Gu,
Pusan, Republic of Korea 609-391

Thesis
T822793 Tsai
c.1 Multilevel security in
data compression and
restricted character set
translation.